

This section uses example designs to show you how to build a system or component. Chapters in this section serve to answer the question, “How do I define systems in SOPC Builder.” This chapter refers to design examples that you can download free from [www.altera.com](http://www.altera.com). Design file hyperlinks are located with individual chapters linked from the Altera website.

This section includes the following chapters:

- [Chapter 9, SOPC Builder Memory Subsystem Development Walkthrough](#)
- [Chapter 10, SOPC Builder Component Development Walkthrough](#)



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter’s revision history.



Most systems generated with SOPC Builder require memory. For example, embedded processor systems require memory for software, while digital signal processing (DSP) systems require memory for data buffers. Many systems use multiple types of memories. For example, a processor-based DSP system can use off-chip SDRAM to store software, and on-chip RAM for fast access to data buffers. You can use SOPC Builder to integrate almost any type of memory into your system.

This chapter uses design examples to describe how to build a memory subsystem as part of a larger system created with SOPC Builder. This chapter focuses on the following kinds of memory most commonly used in SOPC Builder systems:

- “On-Chip RAM and ROM” on page 9–6
- “EPCS Serial Configuration Device” on page 9–9
- “SDR SDRAM” on page 9–11
- “DDR SDRAM” on page 9–14
- “DDR2 SDRAM” on page 9–14
- “Off-Chip SRAM and Flash Memory” on page 9–15

This chapter assumes that you are familiar with the following task and concepts:

- Creating FPGA designs and making pin assignments with the Quartus® II software. For details, refer to the *Introduction to the Quartus II Software* manual.
- Building simple systems with SOPC Builder. For details, refer to the *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon® interfaces. You do not need extensive knowledge of the Avalon interfaces, such as transfer types or signal timing. However, to create your own custom memory subsystem with external memories, you need to understand the Avalon Memory-Mapped (Avalon-MM) interface. For details, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook* and the *Avalon Interface Specifications*.



Refer to the *Memory System Design* chapter in the *Embedded Design Handbook* for additional information on the efficient use of memories in SOPC Builder systems.

## Example Design

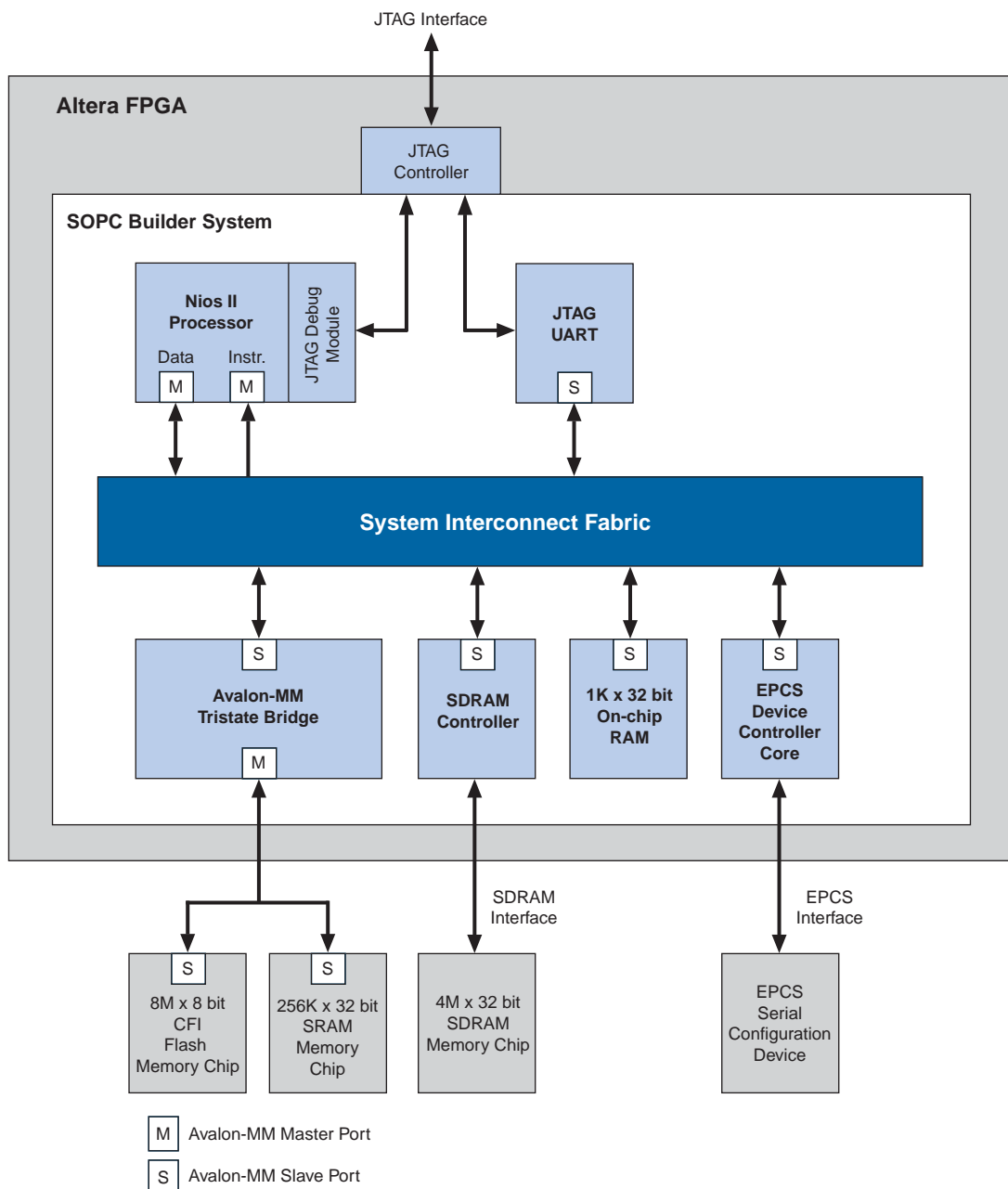
This chapter demonstrates the process for building a system that contains one of each type of memory as shown in [Figure 9–1](#). Each section of the chapter builds on previous sections, culminating in a complete system.

By following the example design in this chapter, you learn how to create a complete customized memory subsystem for your system or design. The memory components in the example design are independent. For a custom system, you only need to instantiate the memories you need. You can also create multiple instantiations of the same type of memory, limited only by on-chip memory resources or FPGA pins to interface with off-chip memory devices.

### Example Design Structure

Figure 9-1 shows a block diagram of the example system.

**Figure 9-1.** Example Design Block Diagram



In [Figure 9-1](#), all blocks shown below the system interconnect fabric comprise the memory subsystem. For demonstration purposes, this system uses a Nios® II processor core to master the memory devices, and a JTAG UART core to communicate with the host PC. However, the memory subsystem could be connected to any master component, located either on-chip or off-chip.

## Example Design Starting Point

The example design consists of the following elements:

- A Quartus II project named **quartus2\_project**. A Block Design File (.bdf) named **oplevel\_design**. **oplevel\_design** is the top-level design file for **quartus2\_project**. **oplevel\_design** instantiates the SOPC Builder system, as well as other pins and modules required to complete the design.
- An SOPC Builder system named **sopc\_memory\_system**. **sopc\_memory\_system** is a subdesign of **oplevel\_design**. **sopc\_memory\_system** instantiates the memory components and other SOPC Builder components required for a functioning SOPC Builder system.

This discussion assumes that the **quartus2\_project** already exists, **sopc\_memory\_system** has been started in SOPC Builder, and the Nios II core and the JTAG UART core are already instantiated. This example design uses the default settings for the Nios II core and the JTAG UART core; these settings do not affect the rest of the memory subsystem.

## Hardware and Software Requirements

To build a memory subsystem similar to the example design in this chapter, you need the following tools:

- Quartus II software version 5.0 or higher—Both Quartus II Web Edition and the fully licensed version support this design flow.
- Nios II Embedded Design Suite (EDS) version 5.0 or higher—Both the evaluation edition and the fully licensed version support this design flow. The Nios II EDS provides the SOPC Builder memory components described in this chapter. It also provides several complete example designs which demonstrate a variety of memory components instantiated in working systems.



The Quartus II Web Edition software and the Nios II EDS, Evaluation Edition are available free for download from the Altera® website. Visit [www.altera.com/download](http://www.altera.com/download). Also, for further reference, see the [Design Examples](#).

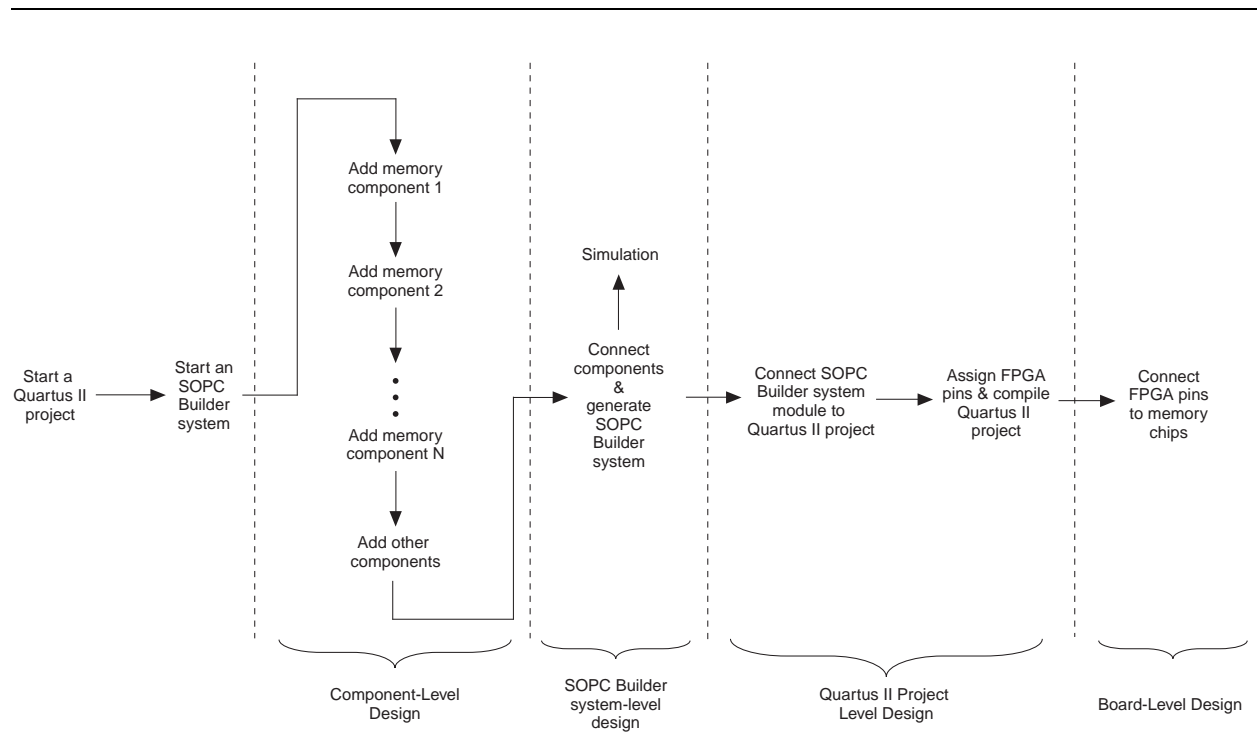
This chapter does not describe downloading and verifying a working system in hardware. Therefore, there are no hardware requirements for the completion of this chapter. However, the example memory subsystem has been tested in hardware.

## Design Flow

This section describes the design flow for building memory subsystems with SOPC Builder, which is similar to other SOPC Builder designs. After starting a Quartus II project and an SOPC Builder system, there are five steps to completing the system, as shown in Figure 9-2:

1. Component-level design in SOPC Builder
2. SOPC Builder system-level design
3. Simulation
4. Quartus II project-level design
5. Board-level design

**Figure 9-2.** Design Flow



### Component-Level Design in SOPC Builder

In this step, you specify which memory components to use and configure each component to meet the needs of the system. All memory components are available from the **Memory and Memory Controllers** category in the list of available components in SOPC Builder.

### SOPC Builder System-Level Design

In this step, you connect components together and configure the SOPC Builder system as a whole. Like the process of adding non-memory SOPC Builder components, you use the **System Contents** tab to do the following:

- Rename the component instance (optional).

- Connect the memory component to masters in the system. Each memory component must be connected to at least one master.
- Assign a base address.
- Assign a clock domain. A memory component can operate on the same or different clock domain as the master(s) that access it.

## Simulation

In this step, you verify the functionality of the SOPC Builder system. For systems with memories, this step depends on simulation models for each of the memory components, in addition to the system testbench generated by SOPC Builder. Refer to “[Simulation Considerations](#)” for more information.

## Quartus II Project-Level Design

In this step, you integrate the SOPC Builder system with the rest of the Quartus II project, which includes connecting the SOPC Builder system to FPGA pins, connecting wiring the SOPC Builder system to other design blocks (such as other HDL modules) in the Quartus II project.



In the example design in this chapter, the SOPC Builder system comprises the entire FPGA design. There are no other design blocks in the Quartus II project.

## Board-Level Design

In this step, you connect the physical FPGA pins to memory devices on the board. If the SOPC Builder system interfaces with off-chip memory devices, you must make board-level design choices.

## Simulation Considerations

SOPC Builder can automatically generate a testbench for RTL simulation of the system using ModelSim®. This testbench instantiates the SOPC Builder system and can also instantiate memory models for external memory components. The testbench is plain text HDL, located at the bottom of the top-level SOPC Builder system HDL design file. To explore the contents of the auto-generated testbench, open the top-level HDL file and search on keyword `test_bench`.



Beginning in ModelSim SE 6.2, design optimization is on by default. Optimization may eliminate design nodes which are referenced in your wave display file. In this case, the you cannot display the waveforms. You can ignore this failure if you want to run an optimized simulation. However, if you want to see the simulation signals, you can disable the optimized compile by setting `VoptFlow = 0` in your `modelsim.ini` file. The `modelsim.ini` is stored in the top-level directory of the ModelSim installation.

### Generic Memory Models

The memory components described in this chapter, except for the SRAM, provide generic simulation models. Therefore, it is very easy to simulate an SOPC Builder system with memory components immediately after generating the system.

The generic memory models store memory initialization files, such as Data (.dat) and Hexadecimal (.hex) files, in a directory named `<Quartus II project directory>/<SOPC Builder system name>_sim`. When generating a new system, SOPC Builder creates empty initialization files. You can manually edit these files to provide custom memory initialization contents for simulation.



For designs that include a Nios II processor, you can create memory initialization files using the Nios II software build tools. For more information, refer to *Creating Memory Initialization Files* in the *Nios II Software Developer's Handbook – Studio Edition*.

### Vendor-Specific Memory Models

You can also manually connect vendor-specific memory models to the SOPC Builder system. In this case, you must manually edit the testbench and connect the vendor memory model. You might also need to edit the vendor memory model slightly for time delays. The SOPC Builder testbench assumes zero delay.

## On-Chip RAM and ROM

Altera FPGAs include on-chip memory blocks that can be used as RAM or ROM in SOPC Builder systems. On-chip memory has the following benefits for SOPC Builder systems:

- On-chip memory has fast access time, compared to off-chip memory.
- SOPC Builder automatically instantiates on-chip memory inside the SOPC Builder system, so you do not have to make any manual connections.
- Certain memory blocks can have initialized contents when the FPGA powers up. This feature is useful, for example, for storing data constants or processor boot code.
- On-chip memories support dual port accesses, allowing two master to access the same memory concurrently.

### Component-Level Design for On-Chip Memory

In SOPC Builder you instantiate on-chip memory by clicking **On-chip Memory (RAM or ROM)** from the list of available components. The configuration wizard for the **On-chip Memory (RAM or ROM)** component has the following options: **Memory type**, **Size**, and **Read latency**.

#### Memory Type

The **Memory type** options define the structure of the on-chip memory:

- **RAM (writable)**—This setting creates a readable and writable memory.
- **ROM (read only)**—This setting creates a read-only memory.

- **Dual-port access**—This setting creates a memory component with two slaves, which allows two masters to access the memory simultaneously.



If two masters access the same address simultaneously in a dual-port memory undefined results will occur. (Concurrent accesses are only a problem for two writes. A read and write to the same location will read out the old data and store the new data.)

- **Block type**—This setting directs the Quartus II software to use a specific type of memory block when fitting the on-chip memory in the FPGA.



The MRAM blocks do not allow the contents to be initialized during power up. The M512s memory type does not support dual-port mode where both ports support both reads and writes.

Because of the constraints on some memory types, it is frequently best to use the **Auto** setting. **Auto** allows the Quartus II software to choose a type and the other settings direct the Quartus II software to select a particular type.

## Size

The **Size** options define the size and width of the memory.

- **Data width**—This setting determines the data width of the memory. The available choices are **8, 16, 32, 64, 128, 256, 512, or 1024** bits. Assign **Data width** to match the width of the master that accesses this memory the most frequently or has the most critical throughput requirements. For example, if you are connecting the on-chip memory to the data master of a Nios II processor, you should set the data width of the on-chip memory to 32 bits, the same as the data-width of the Nios II data master. Otherwise, the access latency could be longer than one cycle because the Avalon interconnect fabric performs width translation.
- **Total memory size**—This setting determines the total size of the on-chip memory block. The total memory size must be less than the available memory in the target FPGA.

## Read Latency

On-chip memory components use synchronous, pipelined Avalon-MM slaves. Pipelined access improves  $f_{MAX}$  performance, but also adds latency cycles when reading the memory. The **Read latency** option allows you to specify either one or two cycles of read latency required to access data. If the **Dual-port access** setting is turned on, you can specify a different read latency for each slave. When you have dual-port memory in your system you can specify different clock frequencies for the ports. You specify this on the **System Contents** tab in SOPC Builder.

## Non-Default Memory Initialization

For ROM memories, you can specify your own initialization file by selecting **Enable non-default initialization file**. This option allows the file you specify to be used to initialize the ROM in place of the default initialization file created by SOPC Builder.

### Enable In-System Memory Content Editor Feature

Enables a JTAG interface used to read and write to the RAM while it is operating. You can use this interface to update or read the contents of the memory from your host PC.



For more information refer to *In-System Updating of Memory and Constants* in volume 3 of the *Quartus II Handbook*.

## SOPC Builder System-Level Design for On-Chip Memory

There are few SOPC Builder system-level design considerations for on-chip memories. See “SOPC Builder System-Level Design” on page 9-4.

When generating a new system, SOPC Builder creates a blank initialization file in the Quartus II project directory for each on-chip memory that can power up with initialized contents. The name of this file is *<name of memory component>.hex*.

## Simulation for On-Chip Memory

At system generation time, SOPC Builder generates a simulation model for the on-chip memory. This model is embedded inside the SOPC Builder system, and there are no user-configurable options for the simulation testbench.

You can provide memory initialization contents for simulation in the file *<Quartus II project directory>/<SOPC Builder system name>\_sim/<Memory component name>.dat*.

## Quartus II Project-Level Design for On-Chip Memory

The on-chip memory is embedded inside the SOPC Builder system, and there are no signals to connect to the Quartus II project.

To provide memory initialization contents, you must fill in the file *<name of memory component>.hex*. The Quartus II software recognizes this file during design compilation and incorporates the contents into the configuration files for the FPGA.



If your design includes a Nios II processor, you can create memory initialization files using the Nios II software build tools. For more information, refer to *Creating Memory Initialization Files* in the *Nios II Software Developer's Handbook – Studio*. For the memory to be initialized, you then must compile the hardware in the Quartus II software for the SRAM Object File (.sof) to pick up the memory initialization files. All memory types with the exception of MRAMs support this feature.

## Board-Level Design for On-Chip Memory

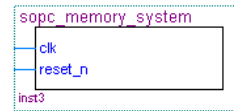
The on-chip memory is embedded inside the SOPC Builder system, and there is nothing to connect at the board level.

## Example Design with On-Chip Memory

This section demonstrates adding a 4 KByte on-chip RAM to the example design. This memory uses a single slave interface with a read latency of one cycle.

For demonstration purposes, [Figure 9-3](#) shows the result of generating the SOPC Builder system at this stage. (In a normal design flow, you generate the system only after adding all system components.)

**Figure 9-3.** SOPC Builder System with On-Chip Memory



Because the on-chip memory is contained entirely within the SOPC Builder system, **sopc\_memory\_system** has no I/O signals associated with **onchip\_ram**. Therefore, you do not need to make any Quartus II project connections or assignments for the on-chip RAM, and there are no board-level considerations.


## EPCS Serial Configuration Device

Many systems use an Altera EPCS serial configuration device to configure the FPGA. Altera provides the EPCS device controller core, which allows SOPC Builder systems to access the memory contents of the EPCS device.

This feature provides flexible design options:

- The FPGA design can reprogram its own configuration memory, providing a mechanism for remote upgrades.
- The FPGA design can use leftover space in the EPCS as nonvolatile storage.

Physically, the EPCS device is a serial flash memory device, which has slow access time. Altera provides software drivers to control the EPCS core for the Nios II processor only.

 For further details about the features and usage of the EPCS device controller core, refer to the *EPCS Device Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

## Component-Level Design for an EPCS Device

In SOPC Builder you instantiate an EPCS controller core by adding an **EPCS Serial Flash Controller** component. There are no settings for this component.

 For details, refer to the *Nios II Flash Programmer User Guide*.

## SOPC Builder System-Level Design for an EPCS Device

There are two SOPC Builder system-level design considerations for EPCS devices:

- Assign a base address.
- Set the IRQ connection to **NC** (no connect). The EPCS controller hardware is capable of generating an IRQ. However, the Nios II driver software does not use this IRQ, and therefore you can leave the IRQ signal disconnected.

There can only be one EPCS controller core per FPGA, and the instance of the core is always named `epcs_controller`.

If you want to store Nios II code in the EPCS memory, point the Nios II reset address at the EPCS controller. Inside the EPCS controller is a bootloader, which Nios II runs after it leaves reset, that copies the code from the EPCS flash into main memory.

## Simulation for an EPCS Device

The EPCS controller core provides a limited simulation model:

- Functional simulation does not include the FPGA configuration process, and therefore the EPCS controller does not model the configuration features.
- The simulation model does not support read and write operations to the flash region of the EPCS device.
- A Nios II processor can boot from the EPCS device in simulation. However, the boot loader code is different during simulation. The EPCS controller boot loader code assumes that all other memory simulation models are initialized, and therefore the boot load process is unnecessary. During simulation, the boot loader simply forces the Nios II processor to jump to start, skipping the boot load process.

Verification in the hardware is the best way to test features related to the EPCS device.

## Quartus II Project-Level Design for an EPCS Device

If you use a device from Cyclone III, Stratix III, or Stratix IV families, you must connect the EPCS pins manually.

For earlier device families, however, the Quartus II software automatically connects the EPCS controller core in the SOPC Builder system to the dedicated configuration pins on the FPGA. This connection is invisible to you. Therefore, there are no EPCS-related signals to connect in the Quartus II project.

## Board-Level Design for an EPCS Device

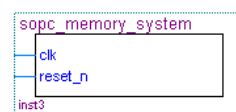
You must connect the EPCS device to the FPGA as described in the Altera *Configuration Handbook*. No other connections are necessary.

## Example Design with an EPCS Device


This section demonstrates adding an EPCS device controller core to the example design.

For demonstration purposes only, [Figure 9-4](#) shows the result of generating the SOPC Builder system at this stage.

**Figure 9-4.** SOPC Builder System with EPCS Device




Because the Quartus II software automatically connects the EPCS controller core to the FPGA pins, the SOPC Builder system has no I/O signals associated with **epcs\_controller**. Therefore, you do not need to make any connections or assignments between the Quartus II project and the EPCS controller core.

 This chapter does not cover the details of configuration using the EPCS device. For further information, refer to the Altera *Configuration Handbook*.

## SDR SDRAM


Altera provides a free SDR SDRAM controller core, which allows you to use inexpensive SDRAM as bulk RAM in your FPGA designs. The SDR SDRAM controller core is necessary, because Avalon-MM signals cannot describe the complex interface on an SDRAM device. The SDR SDRAM controller acts as a bridge between the system interconnect fabric and the pins on an SDRAM device. The SDR SDRAM controller can operate in excess of 100 MHz.

SDR SDRAM is a single data rate SDR SDRAM. Synchronous design allows precise cycle control. With the use of system clock, I/O transactions are possible on every clock cycle. Operating over a range of frequencies, programmable latencies allow the same device to be useful for a variety of high bandwidth, high performance memory system applications.

 For further details about the features and usage of the SDR SDRAM controller core, refer to the *SDR-SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

## Component-Level Design for SDRAM

The choice of SDRAM device(s) and the configuration of the device(s) on the board heavily influence the component-level design for the SDRAM controller. Typically, the component-level design task involves parameterizing the SDRAM controller core to match the SDRAM device(s) on the board. You must specify the structure (address width, data width, number of devices, number of banks, and so on) and the timing specifications of the device(s) on the board.

 For complete details about configuration options for the SDRAM controller core, refer to the *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*.


## SOPC Builder System-Level Design for SDRAM

You can select the SDRAM controller in the SOPC Builder **System Contents** tab. Like the on-chip memory, there are few SOPC Builder system-level design considerations for SDRAM. Refer to “*SOPC Builder System-Level Design*” on page 9-4.

## Simulation for SDRAM

At system generation time, SOPC Builder can generate a generic SDRAM simulation model and include the model in the system testbench. To use the generic SDRAM simulation model, you must turn on a setting in the SDRAM controller configuration wizard. You can provide memory initialization contents for simulation in the file `<Quartus II project directory>/<SOPC Builder system name>_sim/<Memory component name>.dat`.

Alternatively, you can provide a specific vendor memory model for the SDRAM. In this case, you must manually wire up the vendor memory model in the system testbench.

 For further details, refer to “Simulation Considerations” on page 9-5 and the *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

## Quartus II Project-Level Design for SDRAM

SOPC Builder generates a SOPC Builder system with top-level I/O signals associated with the SDRAM controller. In the Quartus II project, you must connect these I/O signals to FPGA pins, which connect to the SDRAM device on the board. In addition, you might have to accommodate clock skew issues.

### Connecting and Assigning the SDRAM-Related Pins

After generating the system with SOPC Builder, you can find the names and directions of the I/O signals in the top-level HDL file for the SOPC Builder system. The file has the name

`<Quartus II project directory>/<SOPC Builder system name>.v` or `<Quartus II project directory>/<SOPC Builder system name>.vhd`. You must connect these signals in the top-level Quartus II design file.

You must assign a pin location for each I/O signal in the top-level Quartus II design to match the target board. Depending on the performance requirements for the design, you might have to assign FPGA pins carefully to achieve the required performance.

### Accommodating Clock Skew

As SDRAM frequency increases, so does the possibility that you must accommodate skew between the SDRAM clock and I/O signals. This issue affects all synchronous memory devices, including SDRAM. To accommodate clock skew, you can instantiate an ALTPLL megafunction in the top-level Quartus II design to create a phase-locked loop (PLL) clock output. You use a phase-shifted PLL output to drive the SDRAM clock and reduce clock-skew issues. The exact settings for the ALTPLL megafunction depend on your target hardware. You must experiment to tune the phase shift to match the board.

 For details, refer to the *ALTPLL Megafunction User Guide*.

## Board-Level Design for SDRAM

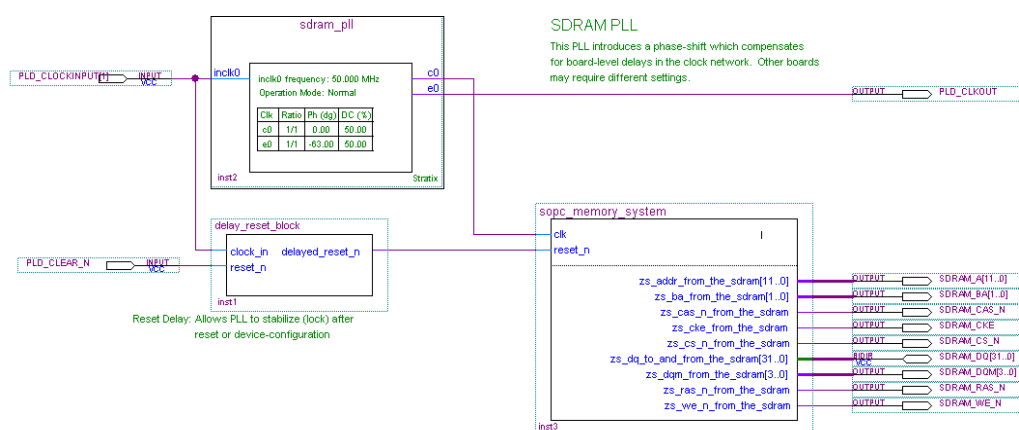
Memory requirements largely dictate the board-level configuration of the SDRAM device or devices. The SDRAM controller core can accommodate various configurations of SDRAM on the board, including multiple banks and multiple devices.

### Example Design with SDR SDRAM

This section demonstrates adding a 16-Mbyte SDRAM device to the example design, using the SDRAM Controller configuration wizard. This SDRAM is a single device with 32-bit data.

For demonstration purposes, [Figure 9-5](#) shows the result of generating the SOPC Builder system at this stage, and connecting it in `toplevel_design.bdf`.

**Figure 9-5.** `toplevel_design.bdf` with SDRAM



After generating the system, the top-level SOPC Builder system file `sopc_memory_system.v` contains the list of SDRAM-related I/O signals that must be connected to FPGA pins. [Example 9-1](#) shows these pins.

#### Example 9-1. I/O Signals Connected to FPGA Pins

```
output [ 11: 0 ] zs_addr_from_the_sdram;
output [  1: 0 ] zs_ba_from_the_sdram;
output          zs_cas_n_from_the_sdram;
output          zs_cke_from_the_sdram;
output          zs_cs_n_from_the_sdram;
inout  [ 31: 0 ] zs_dq_to_and_from_the_sdram;
output [  3: 0 ] zs_dqm_from_the_sdram;
output          zs_ras_n_from_the_sdram;
output          zs_we_n_from_the_sdram;
```

As shown in [Figure 9-5](#), `toplevel_design.bdf` uses an instance of `sdram_pll` to phase shift the SDRAM clock by  $-63$  degrees. (Degrees are relative to clock frequency. If you change the clock speed you must change the phase shift. You should parameterize the PLL with  $-3.5$  ns, because the compensation is for the round-trip delays and clock to I/O delays.)

`toplevel_design.bdf` also uses a subdesign `delay_reset_block` to insert a delay on the `reset_n` signal for the SOPC Builder system. This delay is necessary to allow the PLL output to stabilize before the SOPC Builder system begins operating.


Figure 9-6 shows pin assignments in the Quartus II Assignment Editor for some of the SDRAM pins. The correct pin assignments depend on the target board.

Figure 9-6. Pin Assignments for SDRAM

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved
188	SDRAM_A[0]	PIN_AE4	7	LVTTL	Column I/O		
189	SDRAM_A[10]	PIN_Y11	7	LVTTL	Column I/O		
190	SDRAM_A[11]	PIN_AB7	7	LVTTL	Column I/O		
191	SDRAM_A[1]	PIN_W12	7	LVTTL	Column I/O	PGM0	
192	SDRAM_A[2]	PIN_AC11	7	LVTTL	Column I/O	nR5	
193	SDRAM_A[3]	PIN_W10	7	LVTTL	Column I/O	RUnLU	
194	SDRAM_A[4]	PIN_AA11	7	LVTTL	Column I/O	PGM1	
195	SDRAM_A[5]	PIN_AC10	7	LVTTL	Column I/O	RDN7	
196	SDRAM_A[6]	PIN_AB11	7	LVTTL	Column I/O	RUP7	
197	SDRAM_A[7]	PIN_AC8	7	LVTTL	Column I/O	FCLK5	
198	SDRAM_A[8]	PIN_AB10	7	LVTTL	Column I/O	FCLK4	
199	SDRAM_A[9]	PIN_V11	7	LVTTL	Column I/O		
200	SDRAM_BA[0]	PIN_AG19	8	LVTTL	Column I/O	DQ6B4	
201	SDRAM_BA[1]	PIN_AF19	8	LVTTL	Column I/O	DQ6B5	
202	SDRAM_CAS_N	PIN_AD18	8	LVTTL	Column I/O	DQ6B2	
203	SDRAM_CKE	PIN_AE18	8	LVTTL	Column I/O	DQ6B1	
204	SDRAM_CS_N	PIN_AG18	8	LVTTL	Column I/O	DQ6B0	
205	SDRAM_DQM[0]	PIN_AE14	7	LVTTL	Column I/O	CLK6n	
206	SDRAM_DQM[1]	PIN_Y13	7	LVTTL	Column I/O	CLK7n	
207	SDRAM_DQM[2]	PIN_AE7	7	LVTTL	Column I/O	DQ51B	
208	SDRAM_DQM[3]	PIN_AG10	7	LVTTL	Column I/O	DQ53B	

## DDR SDRAM

You can use double-data rate (DDR) SDRAM devices for a broad range of applications, such as embedded processor systems, image processing, storage, communications, and networking. In addition, the universal adoption of DDR SDRAM in PCs makes DDR SDRAM memory a solution for high-bandwidth applications. DDR SDRAM is a  $<2n>$  prefetch architecture where the internal data bus is twice the width of the external data bus and data transfers occur on both clock edges. It uses a strobe, DQS, which is associated with a group of data pins (DQ) for read and write operations. Both the DQS and DQ ports are bidirectional. Address ports are shared for write and read operations.

 Refer to the DDR SDRAM literature on the Altera website for further details on the use of DDR SDRAM memory, including *AN 517: Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder*.

## DDR2 SDRAM

Double-data rate DDR2 SDRAM is the second generation of double-data rate DDR SDRAM technology, with features such as lower power consumption, higher data bandwidth, enhanced signal quality, and on-die termination. DDR2 SDRAM brings higher memory performance to a broad range of applications, such as PCs, embedded processor systems, image processing, storage, communications, and networking. It is a  $<4n>$  pre-fetch architecture with two data transfers per clock cycle. The memory uses a strobe (DQS) associated with a group of data pins (DQ) for read and write operations. Both the DQ and DQS ports are bidirectional. Address ports are shared for write and read operations.

- For more information refer to the *DDR and DDR2 SDRAM Controller Compiler User Guide*, the *DDR2 SDRAM High-Performance Controller User Guide*, and *AN 517: Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder*.

## Off-Chip SRAM and Flash Memory

SOPC Builder systems can directly access many off-chip RAM and ROM devices, without a controller core to drive the off-chip memory. Avalon-MM signals can describe the interfaces on many standard memories, such as SRAM and flash memory. I/O signals on the SOPC Builder system can connect directly to the memory device.

While off-chip memory usually has slower access time than on-chip memory, off-chip memory provides the following benefits:

- Off-chip memory cost-per-bit is less expensive than on-chip memory resources.
- The size of off-chip memory is bounded only by the 32-bit Avalon-MM address space.
- Off-chip ROM, such as flash memory, can be used for bulk storage of nonvolatile data.
- Multiple off-chip RAM and ROM memories can share address and data pins to conserve FPGA I/O resources at the expense of throughput.

Adding off-chip memories to an SOPC Builder system also requires the **Avalon-MM Tristate Bridge** component.

## Component-Level Design for SRAM and Flash Memory

There are several ways to instantiate an interface to an off-chip memory device:

- For common flash interface (CFI) flash memory devices, add the **Flash Memory (Common Flash Interface)** component in SOPC Builder.
- For Altera development boards, Altera provides SOPC Builder components that interface to the specific devices on each development board. For example, the Nios II EDS includes the components **Cypress CY7C1380C SSRAM** and **IDT71V416 SRAM**, which appear on Nios II development boards.

- For further details about the features and usage of the SSRAM controller core, refer to the *Nios Development Board Cyclone II Edition Reference Manual* or *Nios Development Board Stratix II Edition*.

- For further details about the features and usage of the SDRAM controller core, refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.

These components make it easy for you to create memory systems targeting Altera development boards. However, these components target only the specific memory device on the board; they do not work for different devices.

- For general memory devices, RAM or ROM, you can create a custom interface to the device with the SOPC Builder component editor. Using the component editor, you define the I/O pins on the memory device and the timing requirements of the pins.

In all cases, you must also instantiate the **Avalon-MM Tristate Bridge** component. Multiple off-chip memories can connect to a single tristate bridge, in order to share pins such as the off-chip address bus.

### Avalon-MM Tristate Bridge

A tristate bridge connects off-chip devices to the system interconnect fabric. The tristate bridge creates I/O signals for the SOPC Builder system, which you must connect to FPGA pins in the top-level Quartus II project.

The tristate bridge creates address and data pins that can be shared by multiple off-chip devices. This feature lets you conserve FPGA pins when connecting the FPGA to multiple devices with mutually exclusive access.

You must use a tristate bridge in either of the following cases:

- The off-chip device has bidirectional data pins.
- Multiple off-chip devices share the address, data, or both address and data buses.

In SOPC Builder, you instantiate a tristate bridge by instantiating the **Avalon-MM Tristate Bridge** component. The Avalon-MM Tristate Bridge configuration wizard has a single option: To register incoming (to the FPGA) signals or not.

- **Registered**—This setting adds registers to all FPGA input pins associated with the tristate bridge (outputs from the memory device).
- **Not Registered**—This setting does not add registers between the memory device output pins and the system interconnect fabric.

The Avalon-MM tristate bridge automatically adds registers to output signals from the tristate bridge to off-chip devices.

Registering the input and output signals shortens the register-to-register delay from the memory device to the FPGA, resulting in higher system  $f_{MAX}$  performance. However, the registers add one additional cycle of latency for Avalon-MM masters accessing memory connected to the tristate bridge in each direction. The registers do not affect the timing (setup, hold, and wait) of the transfers from the perspective of the memory device.




For details about the Avalon-MM tristate interface, refer to the [Avalon Interface Specifications](#).

### Flash Memory

In SOPC Builder, you instantiate an interface to CFI flash memory by adding a **Flash Memory (Common Flash Interface)** component. If the flash memory is not CFI compliant, you must create a custom interface to the device with the SOPC Builder component editor.

The choice of flash devices and the configuration of the devices on the board help determine the component-level design for the flash memory configuration wizard. Typically, the component-level design task involves parameterizing the flash memory interface to match the devices on the board. Using the Flash Memory (Common Flash Interface) configuration wizard, you must specify the structure (address width and data width) and the timing specifications of the flash memory devices.

 For details about features and usage, refer to the *Common Flash Interface Controller Core* chapter in volume 5 of the *Quartus II Handbook*.


For an example of instantiating the Flash Memory (Common Flash Interface) component in an SOPC Builder system, see “*Example Design with SRAM and Flash Memory*” on page 9–20.

### SRAM

To instantiate an interface to off-chip SRAM:

1. Create a new component with the SOPC Builder component editor that defines the interface.
2. Instantiate the new interface component in the SOPC Builder system.

The choice of RAM devices and the configuration of the devices on the board determine how you create the interface component. The component-level design task involves entering parameters into the component editor to match the devices on the board.

 For details about using the component editor, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

## SOPC Builder System-Level Design for SRAM and Flash Memory

In the SOPC Builder **System Contents** tab, the Avalon-MM tristate bridge has two ports:

- Avalon-MM slave—This port faces the on-chip logic in the SOPC Builder system. You connect this slave to on-chip masters in the system.
- Avalon-MM tristate master—This port faces the off-chip memory devices. You connect this master to the Avalon-MM tristate slaves on the interface components for off-chip memories.

You assign a clock to the Avalon-MM tristate bridge that determines the Avalon-MM clock cycle time for off-chip devices connected to the tristate bridge.

You must assign base addresses to each off-chip memory. The Avalon-MM tristate bridge does not have an address; it passes unmodified addresses from on-chip masters to off-chip slaves.

## Simulation for SRAM and Flash Memory

The SOPC Builder output for simulation depends on the type of memory components in the system:

- **Flash Memory (Common Flash Interface) component**—This component provides a generic simulation model. You can provide memory initialization contents for simulation in the file `<Quartus II project directory>/<SOPC Builder system name>_sim/<Flash memory component name>.dat`.
- **Custom memory interface created with the component editor**—In this case, you must manually connect the vendor simulation model to the system testbench. SOPC Builder does not automatically connect simulation models for custom memory components to the SOPC Builder system.
- **Altera-provided interfaces to memory devices**—Altera provides simulation models for these interface components. You can provide memory initialization contents for simulation in the file `<Quartus II project directory>/<SOPC Builder system name>_sim/<Memory component name>.dat`. Alternately, you can provide a specific vendor simulation model for the memory. In this case, you must manually wire up the vendor memory model in the system testbench.

For further details, see [“Simulation Considerations” on page 9-5](#).

## Quartus II Project-Level Design for SRAM and Flash Memory

SOPC Builder generates an SOPC Builder system with top-level I/O signals associated with the tristate bridge and the memory interface components. In the Quartus II project, you must connect the I/O signals to FPGA pins, which connect to the memory devices on the board.

After generating the system with SOPC Builder, you can find the names and directions of the I/O signals in the top-level HDL file for the SOPC Builder system. The file has the name `<Quartus II project directory>/<SOPC Builder system name>.v` or `<Quartus II project directory>/<SOPC Builder system name>.vhd`. You must connect these signals in the top-level Quartus II design file.

You must assign a pin location for each I/O signal in the top-level Quartus II design to match the target board. Depending on the performance requirements for the design, you might have to assign FPGA pins carefully to achieve timing.

SOPC Builder inserts synthesis directives in the top-level SOPC Builder system HDL to assist the Quartus II fitter with signals that interface with off-chip devices.

[Example 9-2](#) illustrates a directive. Using `FAST_OUTPUT_REGISTER=ON` places the output register in the IO block, reducing the off-chip delay.



For more information about improving IO timing refer to the I/O Specifications section in [The Quartus II TimeQuest Timing Analyzer](#) chapter in volume 3 of the [Quartus II Handbook](#) and the [Assignment Editor](#) chapter in volume 2 of the [Quartus II Handbook](#).

### Example 9-2. Synthesis Directive

---

```
reg [ 22: 0 ] tri_state_bridge_address /* synthesis
ALTERA_ATTRIBUTE = "FAST_OUTPUT_REGISTER=ON" */;
```

---

## Board-Level Design for SRAM and Flash Memory

Memory requirements determine the board-level configuration of the SRAM and flash memory device or devices. You can lay out memory devices in any configuration, as long as the resulting interface can be described with Avalon-MM signals.



Special consideration is required when connecting the Avalon-MM address signal to the address pins on the memory devices.

The SOPC Builder system presents the smallest number of address lines required to access the largest off-chip memory, which is usually less than 32 address bits. Not all memory devices connect to all address lines.

### Aligning the Least-Significant Address Bits

The Avalon-MM tristate address signal always presents a byte address. Each address location in many memory devices contains more than one byte of data. In this case, the memory device must ignore one or more of the least-significant Avalon-MM address lines. For example, a 16-bit memory device must ignore Avalon-MM address[0] (which is a byte address), and connect Avalon-MM address[1] to the least-significant address line.

Table 9-1 shows the relationship between Avalon-MM address lines and off-chip address pins for all possible Avalon-MM data widths.

**Table 9-1.** Connecting the Least-Significant Avalon-MM Address Line

Avalon-MM Address Line	Address Line Connecting to Memory Device				
	8-bit Memory	16-bit Memory	32-bit Memory	64-bit Memory	128-bit Memory
address[0]	A0	No connect	No connect	No connect	No connect
address[1]	A1	A0	No connect	No connect	No connect
address[2]	A2	A1	A0	No connect	No connect
address[3]	A3	A2	A1	A0	No connect
address[4]	A4	A3	A2	A1	A0
address[5]	A5	A4	A3	A2	A1
address[6]	A6	A5	A4	A3	A2
address[7]	A7	A6	A5	A4	A3
address[8]	A8	A7	A6	A5	A4
address[9]	A9	A8	A7	A6	A5
address[10]	A10	A9	A8	A7	A6
...	...	...	...	...	...

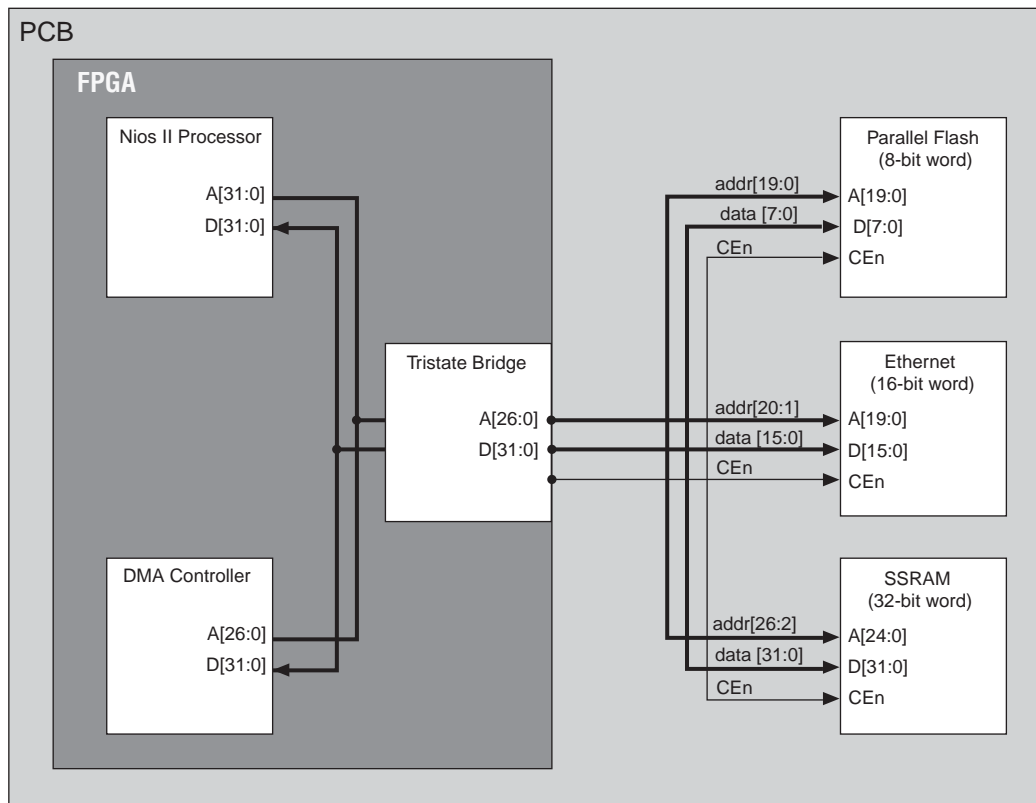


You must ensure that the address bits are properly assigned when mixed width components are connecting to the tristate bridge. Failing to ensure that the components are properly aligned may result in a board respin.

### Aligning the Most-Significant Address Bits

The Avalon-MM address signal contains enough address lines for the largest memory connected to the tristate bridge. Smaller off-chip memories might not use all of the most-significant address lines as [Figure 9-7](#) illustrates.

**Figure 9-7.** Connecting a Tristate Bridge to Components with Different Address Widths and Word Sizes



### Example Design with SRAM and Flash Memory

This section demonstrates adding a 1-MByte SRAM and an 8-MByte flash memory to the example design. These memory devices connect to the system interconnect fabric through an Avalon-MM tristate bridge.

#### Adding the Avalon-MM Tristate Bridge

In the **Avalon-MM Tristate Bridge** configuration wizard, turn on the **Registered inputs and outputs** option to maximize system  $f_{MAX}$ , which increases the read latency by two for both the SRAM and flash memory.

#### Adding the Flash Memory Interface

The flash memory is  $8M \times 8$ -bit, which requires 23 address bits and 8 data bits. [Table 9-2](#) gives the **Flash Memory (Common Flash Interface)** settings for the example design.

**Table 9-2.** Flash Memory Interface (CFI)

Parameter	Value
<b>Attributes</b>	
Presets	AMD29LV065D12R
Address Width (bits)	23
Data Width (bits)	8
<b>Timing</b>	
Setup	40
Wait	160
Hold	40
Units	ns

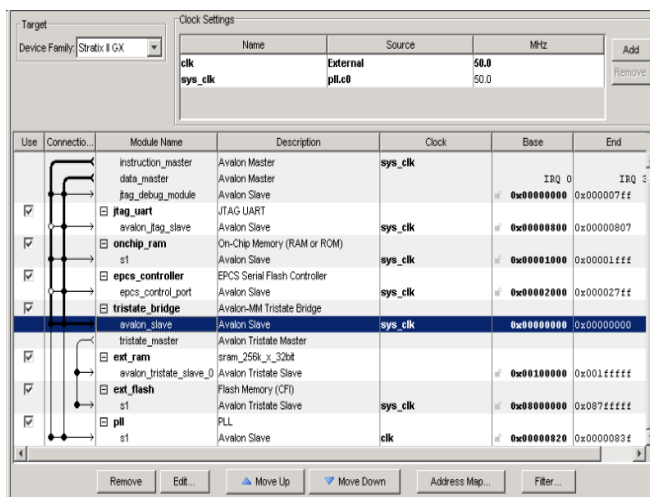
### Adding the SRAM Interface

The SRAM device is 256K × 32-bit, which requires 18 word address bits and 32 data bits. The example design uses a custom memory interface created with the SOPC Builder component editor.

### SOPC Builder System Contents Tab

Figure 9-8 shows the SOPC Builder system after adding the Tristate bridge and memory interface components, and configuring them appropriately on the **System Contents** tab. Figure 9-8 represents the complete example design in SOPC Builder.

**Figure 9-8.** SOPC Builder System with SRAM and Flash Memory



After generating the system, the top-level SOPC Builder system file `sopc_memory_system.v` contains the list of I/O signals for SRAM and flash memory that must be connected to FPGA pins, as shown in Example 9-3.

**Example 9-3.** I/O Signals for SRAM and Flash Memory

```

output      address_to_the_ext_flash[ 23..0];
output      address_to_the_ext_ram[ 19..0];
output      be_n_to_the_ext_ram[ 3..0];
output      read_n_to_the_ext_flash;
output      read_n_to_the_ext_ram;
output      read_n_to_the_ext_ram;
output      select_n_to_the_ext_flash;
output      select_n_to_the_ext_ram;
bidirectional tristate_bridge_data [ 31..0]
output      write_n_to_the_ext_flash;
output      write_n_to_the_ext_ram;

```

The Avalon-MM tristate bridge signals that can be shared are named after the instance of the tristate bridge component, such as `tri_state_bridge_data[31:0]`.

**Connecting and Assigning Pins in the Quartus II Project**

Figure 9-9 shows the result of generating the SOPC Builder system for the complete example design.

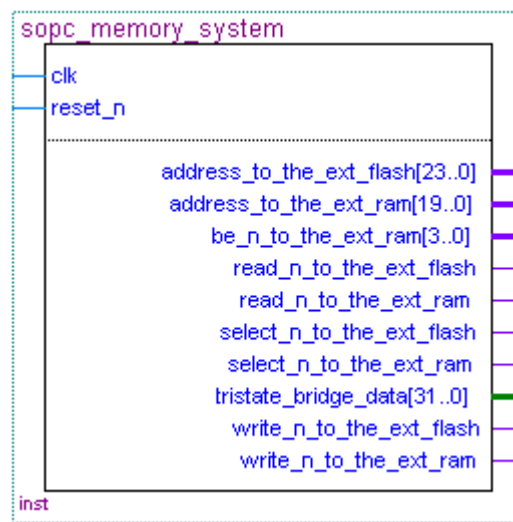
**Figure 9-9.** Top Level System with SRAM and Flash Memory

Figure 9-10 shows the pin assignments in the Quartus II Assignment Editor for some of the SRAM and flash memory pins. The correct pin assignments depend on the target board.

**Figure 9-10.** Pin Assignments for SRAM and Flash Memory

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reset
243	SRAM_BE_N[0]	PIN_M18	3	LVTTTL	Column I/O		
244	SRAM_BE_N[1]	PIN_F17	3	LVTTTL	Column I/O		
245	SRAM_BE_N[2]	PIN_J18	3	LVTTTL	Column I/O	RUP3	
246	SRAM_BE_N[3]	PIN_L17	3	LVTTTL	Column I/O	CLK15n	
247	SRAM_CS_N	PIN_B24	3	LVTTTL	Column I/O	DQ9T4	
248	SRAM_OE_N	PIN_B26	3	LVTTTL	Column I/O	DQ9T7	
249	SRAM_WE_N	PIN_C24	3	LVTTTL	Column I/O	DQ59T	

## Connecting FPGA Pins to Devices on the Board

Table 9-3 shows the mapping between the Avalon-MM address lines and the address pins on the SRAM and flash memory devices.

**Table 9-3.** FPGA Connections to SRAM and Flash Memory

Avalon-MM Address Line	Flash Address (8M × 8-bit Data)	SRAM Address (256K × 32-bit data)
tri_state_bridge_address[0]	A0	No connect
tri_state_bridge_address[1]	A1	No connect
tri_state_bridge_address[2]	A2	A0
tri_state_bridge_address[3]	A3	A1
tri_state_bridge_address[4]	A4	A2
tri_state_bridge_address[5]	A5	A3
tri_state_bridge_address[6]	A6	A4
tri_state_bridge_address[7]	A7	A5
tri_state_bridge_address[8]	A8	A6
tri_state_bridge_address[9]	A9	A7
tri_state_bridge_address[10]	A10	A8
tri_state_bridge_address[11]	A11	A9
tri_state_bridge_address[12]	A12	A10
tri_state_bridge_address[13]	A13	A11
tri_state_bridge_address[14]	A14	A12
tri_state_bridge_address[15]	A15	A13
tri_state_bridge_address[16]	A16	A16
tri_state_bridge_address[17]	A17	A15
tri_state_bridge_address[18]	A18	A16
tri_state_bridge_address[19]	A19	A17
tri_state_bridge_address[20]	A20	No connect
tri_state_bridge_address[21]	A21	No connect
tri_state_bridge_address[22]	A22	No connect

## Document Revision History

Table 9-4 shows the revision history for this chapter.

**Table 9-4.** Document Revision History(Sheet 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2009, v9.1.0	No changes from previous release.	—
March 2009, v9.0.0	Minor updates to clarify text.	—
November 2008, v8.1.1	<ul style="list-style-type: none"> <li>■ Removed private comments</li> </ul>	—
November 2008, v8.1.0	<ul style="list-style-type: none"> <li>■ Added text explaining that starting in 6.2, ModelSim turns the VoptFlow option on by default which may optimize away nodes included in preset wave file.</li> <li>■ Changed page size to 8.5 x 11 inches</li> </ul>	—
May 2008, v8.0.0	<ul style="list-style-type: none"> <li>■ Chapter renumbered from 8 to 9.</li> <li>■ Added brief new sections referencing DDR-2 and PFLs.</li> <li>■ Updated references to Avalon Interface Specifications.</li> <li>■ Updated Figures 9-1, 9-14, 9-15, 9-16, and 9-19 with new art.</li> </ul>	—
October 2007, v7.2.0	<ul style="list-style-type: none"> <li>■ Corrected Figure 9-9 to show flash memory changed example to use a PLL that is part of the SOPC Builder system, rather than a Quartus II component. Added section showing parameterization of PLL.</li> </ul>	—
May 2007, v7.1.0	<ul style="list-style-type: none"> <li>■ Chapter 8 was previously chapter 9.</li> <li>■ Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon switch fabric” term to “system interconnect fabric.” Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.”</li> <li>■ Added section on Non-Default Memory Initialization.</li> <li>■ On-chip Memory size, first parameter changed from Memory Width to Data Width and widths of 256, 512 and 1024 were added.</li> <li>■ Corrected figure 8-18.</li> <li>■ Added links to all referenced documents.</li> <li>■ Removed discussions of reference designators for components because they are no longer required by SOPC Builder.</li> <li>■ Removed unnecessary screenshots.</li> </ul>	Updated to reflect changes to SOPC Builder for 7.1.0. SOPC Builder and improve readability.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	No change from previous release.	—
May 2006, v6.0.0	Chapter 9 was previously chapter 8. No change to content.	—

**Table 9-4.** Document Revision History(Sheet 2 of 2)

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
October 2005, v5.1.0	Chapter 8 was previously chapter 6. No change to content.	—
May 2005, v5.0.0	Initial release.	—



This chapter describes the parts of a custom SOPC Builder component and guides you through the process of creating an example custom component, integrating it into a system, and testing it in hardware.

This chapter is divided into the following sections:

- [“Component Development Flow” on page 10–2.](#)
- [“Design Example: Checksum Hardware Accelerator” on page 10–4.](#) This design example shows you how to develop a component with both Avalon® Memory-Mapped (Avalon-MM) master and slaves.
- [“Sharing Components” on page 10–6.](#) This section shows you how to use components in other systems, or share them with other designers.
- [“System Information Files \(.sopcinfo\)” on page 10–7.](#)

## SOPC Builder Components and the Component Editor

An SOPC Builder component is usually composed of the following four types of files:

- HDL files—define the component’s functionality as hardware.
- Hardware Component Description File (`_hw.tcl`)—describes the SOPC Builder related characteristics, such as interface behaviors. This file is created by the component editor.
- C-language files—define the component register map and driver software to allow programs to control the component.
- Software Component Description File (`_sw.tcl`) file—used by the software build tools to use and compile the component driver code.

The component editor guides you through the creation of your component. You can then instantiate the component in an SOPC Builder system and make connections in the same manner as other SOPC Builder components. You can also share your component with other designers.

For information about creating the `_sw.tcl` file, see the [Developing Device Drivers for the Hardware Abstraction Layer](#) chapter in the *Nios II Software Developer’s Handbook*.

## Prerequisites

This chapter assumes that you are familiar with the following:

- Building systems with SOPC Builder. For details, refer to the [Introduction to SOPC Builder](#) chapter in volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, refer to the [SOPC Builder Components](#) chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon-MM interface.

## Hardware and Software Requirements

To use the design example in this chapter, in addition to the current version of the Quartus II software and Nios II Embedded Design Suite, you must have the following:

- Design files for the example design—A hyperlink to the design files appears next to the chapter, *SOPC Builder Component Development Walkthrough*, on the [SOPC Builder literature page](#).
- Nios development board and an Altera® USB-Blaster™ download cable—You can use either of the following Nios development boards:
  - Stratix® II Edition, RoHS compliant version
  - Cyclone® II Edition

If you do not have a development board, you can follow the hardware development steps. You cannot download the complete system without a working board, but you can simulate the system.



You can download the Quartus II Web Edition software and the Nios II EDS, Evaluation Edition for free from the Altera Download Center at [www.altera.com](http://www.altera.com).

## Component Development Flow

This section provides an overview of the development process for SOPC Builder components.

### Typical Design Steps

A typical development sequence for an SOPC Builder component includes the following items:

1. Specification and definition.
  - a. Define the functionality of the component.
  - b. Determine component interfaces, such as Avalon Memory-Mapped (Avalon-MM), Avalon Streaming (Avalon-ST), interrupt, or other interfaces.
  - c. Determine the component clocking requirements; what interfaces are synchronous to what clock inputs.
  - d. If you want a microprocessor to control the component, determine the interface to software, such as the register map.
2. Implement the component in VHDL or Verilog HDL.

3. Import the component into SOPC Builder.
  - a. Use the component editor to create a `_hw.tcl` file that describes the component.
  - b. Instantiate the component into an SOPC Builder system.

When importing an HDL file using the component editor, any parameter definitions that are dependent upon other defined parameters cause an error. [Example 10-1](#) illustrates the declaration of a `DEPTH` parameter which is legal Verilog HDL syntax in the Quartus II software, but causes an error in the component editor syntax checker.

---

**Example 10-1. DEPTH Parameter**

---

```
parameter WIDTH = 32;  
parameter DEPTH = ((WIDTH == 32) ? 8 : 16);
```

---

To avoid this error, use a `localparam` for the dependent parameter instead, as shown in [Example 10-2](#).


---


**Example 10-2. localparam Parameter**

---

```
parameter WIDTH = 32;  
localparam DEPTH = ((WIDTH == 32)?8:16);
```

---

 SOPC Builder only supports the VHDL port types `std_logic` and `std_logic_vector`.

4. Develop the software driver, which can occur in parallel with the hardware implementation. Create the component's driver, including a C header file that defines the hardware-level register map for software.
  -  For further details, see the *Nios II Software Developer's Handbook*.
5. Perform in-system testing, such as the following:
  - a. Test register-level accesses to the component in hardware or simulation using a microprocessor, such as the Nios II processor.
  - b. Performance benchmarking.

## Hardware Design

As with any logic design process, the development of SOPC Builder component hardware begins after the specification phase. Creating the HDL design is often an iterative process, as you write and verify the HDL logic against the specification.

The architecture of a typical component consists of the following functional blocks:

- **Task logic**—Implements the component's fundamental function. The task logic is design dependent.
- **Interface logic**—Provides a standard way of providing data to or getting data from the components and of controlling the functioning of the components.


 For further details, refer to the *Avalon Interface Specifications*.

Figure 10-1 shows the top-level blocks of a checksum component, which includes both Avalon-MM master and slaves.



The work flow for developing SOPC Builder hardware, including how to decide upon and implement the register map, is described in the *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*. Also, guidelines for developing device drivers is described in the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

## Design Example: Checksum Hardware Accelerator

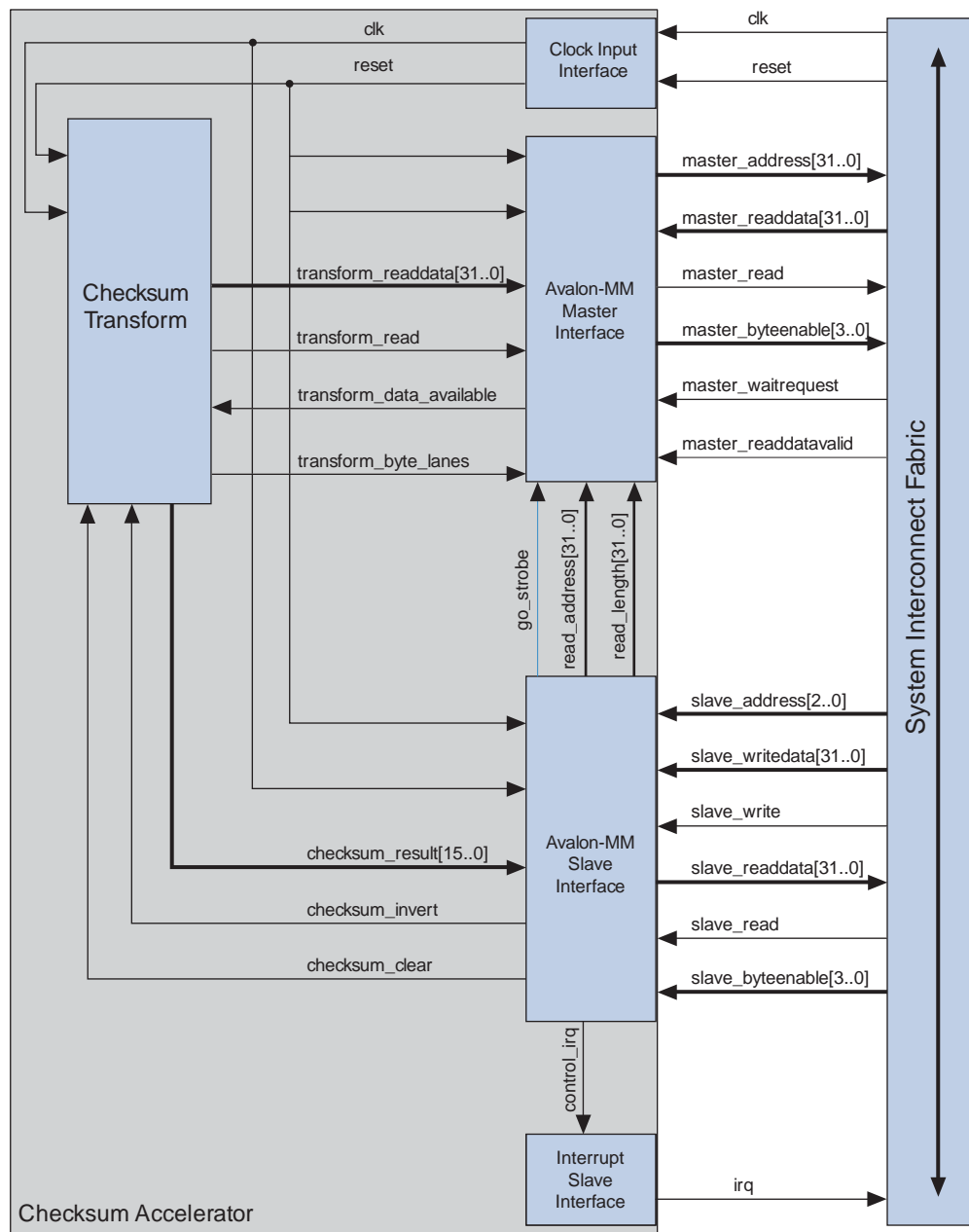
Altera has provided a checksum hardware accelerator design example to demonstrate the steps to create a component and instantiate it in a system. This design example is available for download from the Altera literature website. Included in the compressed download file is a **readme.pdf** that describes how to create and compile the hardware design, and describes how to use the checksum hardware accelerator in your design.

You can use the checksum algorithm in network applications where data integrity must be inspected by the receiving device. The checksum algorithm accumulates data with end-round-carry summation, which means that the carry bit from the accumulator is added to the least significant bit of the next input. After the data is accumulated, you can use the result to verify the data integrity of the data buffer. Because the checksum algorithm operates over a data buffer, you can implement it more efficiently with a pipelined read master. A pipelined read master continuously posts read transactions minimizing the effects of the memory read latency. The checksum accelerator can read data and calculate the checksum result every clock cycle, which you cannot do with a general purpose processor.

The checksum hardware accelerator requires information from a host processor such as the buffer base address, buffer length, and various control signals. As a result, the hardware accelerator exposes an Avalon-MM slave interface so that a host processor can control the read master operation. The host processor also accesses the checksum result from the slave interface. Each piece of information sent or read by the host processor is accessed separately in the register file implemented with the slave interface. For example, the status and control signals are implemented as separate registers because they contain information used for different purposes and have different access capabilities.

Hardware accelerators can operate in parallel with a host processor; consequently, adding an interrupt sender interface to the hardware accelerator increases system performance. While the accelerator is operating on a buffer, the host processor can perform other tasks such as preparing another buffer for transmission. The interrupt is asserted after the buffer checksum is calculated. The host processor can be interrupted by the hardware accelerator to notify it that a checksum result has been calculated. The host processor can then read the checksum value and clear the interrupt by writing to the status register via the accelerator slave interface.

Figure 10-1. Checksum Component with Avalon-MM Master and Slaves



## Software Design


If you want a microprocessor to control your component, you must provide software files that define the software view of the component. At a minimum, you must define the register map for each Avalon-MM slave that is accessible to a processor.



In the example checksum project, you can view an example of a software driver in the directory `<projectdir>/ip/checksum_accelerator`, which is the top level folder of the hardware and software for the custom checksum block.

Software drivers abstract hardware details of the component so that software can access the component at a high level. The driver functions provide the software an API to access the hardware. The software requirements vary according to the needs of the component. The most common types of routines initialize the hardware, read data, and write data.

When developing software drivers, you should review the software files provided for other ready-made components. The IP installer provides many components you can use as reference. You can also view the `<Nios II EDS install path>/components/` directory for examples.


 For details about writing drivers for the Nios II hardware abstraction layer (HAL), refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

## Verifying the Component

You can verify the component in incremental stages, as you complete more of the design. You should first verify the hardware logic as a unit (which might consist of multiple smaller stages of verification) and later verify the component in a system.

### System Console


The system console is an interactive Tcl console available from within SOPC Builder that provides you with read and write access to the debugging capabilities that are available in your FPGA logic. You can use the system console to control and query the state of the Nios II processor, issue Avalon transactions, board bring-up, and access either JTAG UARTs or system level debug (SLD) nodes.

 For further details, refer to the *System Console User Guide*.

### System-Level Verification

After you package a `_hw.tcl` file with the component editor, you can instantiate the component in a system and verify the functionality of the overall SOPC Builder system.

SOPC Builder provides support for system-level verification for HDL simulators such as ModelSim®. SOPC Builder automatically produces a test bench for system-level verification.

 You can include a Nios II processor in your system to enhance simulation capabilities during the verification phase. Even if your component has no relationship to the Nios II processor, the auto-generated ModelSim simulation environment provides an easy-to-use starting point.

## Sharing Components

When you create a component, component editor saves the `_hw.tcl` file in the same directory as the top-level HDL file. Where appropriate, files referenced by the `_hw.tcl` file are specified relative to the `_hw.tcl` file itself, so the files can easily be moved and copied. To share a component, include it in your IP library.

For more information about including components in an IP library refer to *Finding Components in SOPC Builder* in *Chapter 4: SOPC Builder Components* in volume 4 of the *Quartus II Handbook*.

## System Information Files (.sopcinfo)

Every time SOPC Builder generates a system, a `<mysystem>.sopcinfo` is also generated, which contains the following information:

- SOPC Builder project, including:
  - Name and tool version
  - HDL language
- Each module instantiated in the system, including:
  - Name and version
  - Where interface information was found on the disk, such as signal names and types, interface properties, and clock domain mapping
  - Parameter names and values
- Each connection, including:
  - Component and interface connections
  - Base address, Avalon-MM interfaces, IRQ number interfaces
  - Memory map as seen by each master in the system



The `.sopcinfo` file is a report file only, and cannot be edited with SOPC Builder.

## Document Revision History

Table 10-1 shows the revision history for this chapter.

**Table 10-1.** Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2009, v9.1.0	<ul style="list-style-type: none"> <li>Added statement that SOPC Builder only supports the VHDL <code>std_logic</code> and <code>std_logic_vector</code> port types.</li> </ul>	
March 2009, v9.0.0	Corrected direction of <code>transform_data_available</code> and <code>transform_byte_lanes</code> signals in Figure 10-1 on page 10-5.	One correction.
November 2008, v8.1.0	<ul style="list-style-type: none"> <li>Added reference to new search path for IP chapter 4 of this volume.</li> <li>Correction direction of signals in Figure 10-1.</li> <li>Changed page size to 8.5 x 11 inches.</li> </ul>	One correction and one change to reflect changes in underlying software.
May 2008, v8.0.0	<ul style="list-style-type: none"> <li>Chapter renumbered from 9 to 10.</li> <li>Removed discussion of the Checksum Design example, which will now be in a <code>readme.pdf</code> file and zipped with the rest of the design files.</li> <li>Deleted references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications.</li> <li>New Figure 9-1 and Table 9-1.</li> <li>New section on <code>.sopcinfo</code> file.</li> </ul>	Deleted example procedure.
October 2007, v7.2.0	<ul style="list-style-type: none"> <li>Updated instructions on how to develop components to match new GUI.</li> </ul>	—
May 2007, v7.1.0	<ul style="list-style-type: none"> <li>Changed example component from a pulse width modulator with that only has an Avalon-MM slave interface to a checksum master that includes both Avalon-MM master and slave interfaces.</li> </ul>	Changed the example design to one with more practical applications. Updated instructions for the 7.1 release.
March 2007, v7.0.0	<ul style="list-style-type: none"> <li>No change from previous release.</li> </ul>	—
November 2006, v6.1.0	<ul style="list-style-type: none"> <li>Chapter 9 was previously chapter 10. No change to content.</li> </ul>	—
May 2006, v6.0.0	<ul style="list-style-type: none"> <li>Chapter 10 was previously chapter 9. No change to content.</li> </ul>	—
October 2005, v5.1.0	<ul style="list-style-type: none"> <li>Chapter 9 was previously chapter 7. No change to content.</li> </ul>	—
August 2005, v5.0.1	<ul style="list-style-type: none"> <li>Corrected Table 7-5.</li> </ul>	—
May 2005, v5.0.0	<ul style="list-style-type: none"> <li>No change from previous release.</li> </ul>	—
February 2005, v1.0	<ul style="list-style-type: none"> <li>Initial release.</li> </ul>	—