Using Tightly Coupled Memory with the Nios II Processor

Tutorial

101 Innovation Drive
San Jose, CA 95134
www.altera.com

TU-N2060305-2.0

Document last updated for Altera Complete Design Suite version: 11.0
Document publication date: July 2011

Subscribe

Contents

1. Using Tightly Coupled Memory with the Nios II Processor

This document describes how to use tightly coupled memory in designs that include a Nios® II processor and discusses some possible applications. It also includes a tutorial that guides you through the process of building a Nios II system with tightly coupled memory.

The Nios II architecture includes tightly coupled master ports that provide guaranteed fixed low-latency access to on-chip memory for performance-critical applications. Tightly coupled master ports can be connected to instruction memory and data memory, to allow fixed low-latency read access to executable code as well as fixed low-latency read or write access to data. Tightly coupled masters are dedicated instruction or data master ports on the Nios II core, which is different from the embedded processor's instruction and data master ports.

This document assumes you are familiar with the Nios II tightly coupled memory. For more information, refer to the *Processor Architecture* chapter in the *Nios II Processor Reference Handbook*.

Reasons for Using Tightly Coupled Memory

You can implement one or more of the following functions or modules using tightly coupled memory to enhance the performance of your system:

- Separate exception stack for use only while handling interrupts
- Fast data buffers
- Fast sections of code
  - Fast interrupt handler
  - Critical loop
- Constant access time that is guaranteed not to have arbitration delays

For programs with modest memory requirements, all of the code and data can be held in a tightly coupled memory pair.

Tradeoffs

There are design tradeoffs when using tightly coupled memory, including the following:

- You must balance the benefits of more general speed enhancement provided by a cache, averaged over time, with the specific dedicated memory block consumed by a tightly coupled memory whose sole purpose is to make a single part of the code faster.
- Software guarantees that performance-critical code or data is located in tightly coupled memory. That particular piece of code or data achieves high performance. Locating the code within tightly coupled memory eliminates cache overhead such as cache flushing, loading, or invalidating.

- You must divide on-chip memory equitably to provide the best overall combination of tightly coupled instruction memory, tightly coupled data memory, instruction cache, and data cache.

Guidelines for Using Tightly Coupled Memory

This section details the guidelines and limitations that should be taken into consideration when designing the hardware and software with tightly coupled memory.

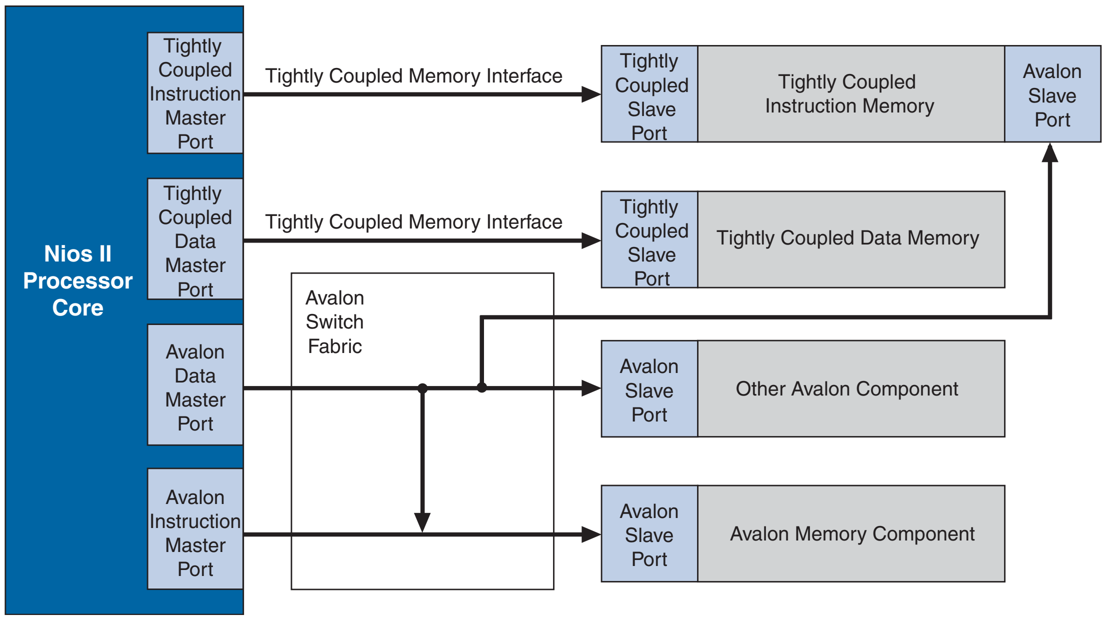
Hardware Guidelines

The following guidelines apply to Nios II hardware designs that include tightly coupled memory:

- Tightly coupled masters are presented as additional master ports on the Nios II processor.
- An On-Chip Memory component is the only memory that can connect to a tightly coupled master port on the Nios II core.
- A tightly coupled master on a processor must connect to exactly one on-chip memory slave port. This slave port cannot be shared by any other master port.
- Each on-chip memory can be connected to only one tightly coupled master even if it is a dual port memory.
- The availability of the data and instruction master ports for the tightly coupled memory is dependent on the type of Nios II core used.
- When using the On-Chip Memory component as a tightly coupled memory for Nios II, you must always create it as a RAM, not as a ROM. Tightly coupled memories configured as ROM would result in failure.
- To conserve the logic elements, use one 2-kilobyte (KB) tightly coupled memory, rather than two 1-KB tightly coupled memories.

Figure 1 is a block diagram of a simple Nios II system that includes tightly coupled memories and other Qsys System Integration Tool components.

Figure 1. Nios II System with Tightly Coupled Instruction and Data Memory


Nios II Processor Core
Tightly Coupled Instruction Master Port
Tightly Coupled Data Master Port
Avalon Data Master Port
Avalon Instruction Master Port
Tightly Coupled Memory Interface
Tightly Coupled Memory Interface
Avalon Switch Fabric
Tightly Coupled Slave Port
Tightly Coupled Instruction Memory
Avalon Slave Port
Tightly Coupled Slave Port
Tightly Coupled Data Memory
Avalon Slave Port
Other Avalon Component
Avalon Slave Port
Avalon Memory Component

Software Guidelines

The following two guidelines apply to Nios II software that uses tightly coupled memory:

- Software accesses tightly coupled memory addresses just like any other addresses.
- Cache operations have no effect when targeting tightly coupled memory.

Locating Functions in Tightly Coupled Memory

Assigning data to a tightly coupled data memory also involves using a section attribute. Alternatively, you can include the tightly coupled memory as a `#define` in the **system.h** file. The name of the memory is followed by `_BASE` and is used as a pointer to reference the tightly coupled data memory.

The software example in this tutorial provides a source code example showing how to locate a particular source code function in a particular linker section. A function is declared to reside within a linker section with the C section attribute in the file **timer_interrupt_latency.h**. This C header file locates `timer_interrupt_latency_irq()` in the `.exceptions` section as follows:

```
externvoid timer_interrupt_latency_irq (void* base, alt_u32 id)__attribute__ ((section
(".exceptions")));
```

The Nios II Software Build Tools (SBT) creates linker sections for each memory module in the system. A source code function can be located within a particular tightly coupled instruction memory simply by assigning that function to the linker section created for that tightly coupled instruction memory.

The Nios II SBT creates additional linker sections with address mappings that are controlled by Qsys. For the `.exceptions` section, the physical address offset and memory module in which to base that linker section is manipulated through Qsys. You locate the `.exceptions` section in a memory module covered by a tightly coupled data memory using the **Exception Vector** field found on the **Core Nios II** tab of the configuration wizard.

For more information about the C section attribute, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter in section 2 of the *Nios II Software Developer's Handbook*.

Tightly Coupled Memory Interface

The term *tightly coupled memory interface* refers to an Avalon®-like interface that connects one master to one slave. Refer to Figure 1. Tightly coupled memory interfaces connect tightly coupled masters to their tightly coupled slaves. Tightly coupled memory interfaces are designed to be connected to one port of an on-chip memory device. These devices are known as "altsyncrams" to Verilog HDL and VHDL designers.

Restrictions

You must observe the following restrictions when designing with tightly coupled memories:

- Tightly coupled slaves must be on-chip memories.
- Only one master and one slave can be connected to a given tightly coupled memory interface, which makes the tightly coupled memory interface a point-to-point connection.
- Tightly coupled slaves have a data width of 32 bits. Tightly coupled memory interfaces do not support dynamic bus sizing.
- Tightly coupled slaves have a read latency of one cycle, a write latency of zero cycles, and no wait states.

When tightly coupled memory is present, the Nios II core decodes addresses internally to determine if the requested instructions or data reside in tightly coupled memory. If the address resides in tightly coupled memory, the Nios II core accesses the instruction or data through the tightly coupled memory interface. Accessing tightly coupled memory bypasses cache memory. The processor core functions as if cache were not present for the address span of the tightly coupled memory. Instructions for managing the cache do not affect the tightly coupled memory, even if the instruction specifies an address in the range occupied by a tightly coupled memory.

Dual Port Memories

Each tightly coupled master connects to one tightly coupled slave over a tightly coupled interface. For this reason, it is helpful to use dual port memories with the tightly coupled instruction master as shown in Figure 1. The tightly coupled instruction master is incapable of performing writes because it accesses code for execution only. Without a second memory port connected to an Avalon Memory-Mapped (Avalon-MM) data master, the system does not have write access to the tightly coupled instruction memory. Without write access, code cannot be downloaded into the tightly coupled memory by the Nios II SBT for Eclipse, which makes development and debugging difficult. Without a second port on the tightly coupled instruction memory, no data master has access to the memory, which means you have no way to view the contents. By making the tightly coupled instruction memory dual port, the embedded processor's data master can be connected to the second port, allowing both reading and writing of data.

Building a Nios II System with Tightly Coupled Memory

This section provides a detailed list of instructions to create a Nios II system in Qsys that uses two tightly coupled memories, one for instruction access and one for data access. These two tightly coupled memories are connected to the Nios II processor as shown in Figure 1 on page 1–3. Additionally, instructions are provided to build a software project to exercise these tightly coupled memories. The output of the software shows that the tightly coupled memories have much faster access times than other on-chip memories.

To build a Nios II system with tightly coupled memory, perform the following steps. These steps are described more fully in the following sections.

1. Modify an existing reference design to include tightly coupled memories.
2. Create the tightly coupled memories in Qsys.
3. Connect the tightly coupled memories to the masters.
4. Position the tightly coupled memories in the Nios II processor's address map.
5. Specify the Nios II exception address to access tightly coupled instruction memory.
6. Add a performance counter.
7. Generate the hardware system.
8. Create a software project to exercise the tightly coupled memories.
9. Execute the software on the new hardware design.
10. Change the Tcl scripts and recompile the design to review how the timer settings work.

Hardware and Software Requirements

The following hardware and software are required to perform this exercise:

- Nios II development tools version 11.0 or later
- Quartus® II software version 11.0 or later
- One of the following Altera development kit boards:
  - Nios II Embedded Evaluation Kit (NEEK), Cyclone® III Edition
  - Embedded Systems Development Kit (ESDK), Cyclone III Edition
  - Stratix® IV GX FPGA Development Kit

Modify the Example Design to Include Tightly Coupled Memories

First, create a new hardware reference design with tightly coupled memories that is based on the Nios II Ethernet standard design example. To create this modified reference design, perform the following steps:

1. Navigate to the Nios II Ethernet Standard Design Example web page and locate the Nios II Ethernet Standard design example **.zip** file that corresponds to your board.
2. Extract the files from the downloaded **.zip** file and copy the **niosii_ethernet_standard_**<*board*> directory to a new directory named **standard_tcm**.
3. In the Windows Start menu, choose **Programs** > **Altera** > **Quartus II** <*version*> to run the Quartus II software.
4. On the File menu, click **Open Project** and browse to the **standard_tcm\niosii_ethernet_standard_**<*board*>**.qpf** project file. Click **Open**.
5. On the Tools menu, click **Qsys**. When prompted, select **eth_std_main_system.qsys** and click **Open** to open the Qsys design.
6. Double-click the **cpu** component in the list of available components on the **System Contents** tab to open the Nios II processor configuration wizard.
7. Ensure that **Nios II/f** is selected on the **Core Nios II** tab.
8. Click the **Caches and Memory Interfaces** tab.
9. Select the number of instruction master ports in the drop-down list next to **Number of tightly coupled instruction master port(s)**. In this example, select **1** port.
10. Change the **Instruction cache** to **4** KB.
11. Select the number of instruction master ports in the drop-down list next to **Number of tightly coupled data master port(s)**. In this example, select **1** port.
12. Change the **Data cache** to **2** KB and **Data Cache Line Size** to **4** Bytes.
13. Click **Finish** to close the Nios II processor configuration wizard.

Two new master ports now appear under the **cpu** component called **tightly_coupled_instruction_master_0** and **tightly_coupled_data_master_0**. These master ports are not yet connected to the slave ports.

Create the Tightly Coupled Memories

In this section you create two types of tightly coupled memories: a tightly coupled instruction memory and a tightly coupled data memory.

1. In the **Component Library** tab, double-click **On-Chip Memory** in the **On-Chip** subfolder of the **Memories and Memory Controllers** folder. The On-Chip Memory configuration wizard appears.
2. To complete the configuration of this memory, specify the settings listed in Table 1.
3. In the **Size** and **Memory** initialization boxes, specify the settings listed in Table 1.

Table 1. On-Chip Memory Default Settings

Properties		Configuration Settings
Memory type	**RAM**	Turn this option on
	Dual-Port Access	Turn this option on
	Read During Write Mode	This option is **DONT_CARE**
	Block type	**Auto**
Size	**Memory Width**	**32**
	Total memory size	**4 KB**
Read latency	**Slave s1**	**1**
	Slave s2	**1**
Memory initialization	**Initialize memory content**	Turn this option on
	Enable non-default initialization file	Leave this option off
	Enable In-System Memory Content Editor feature	Leave this option off

4. Click **Finish** to close the configuration wizard.
5. Click the **System Contents** tab and scroll down to the **onchip_memory2_0** component.

CAUTION You must name the components exactly as they appear in this tutorial. If your component names differ from the names printed here, the software example will not work.

6. Right-click **onchip_memory2_0** and rename the component to **tightly_coupled_instruction_memory**.
7. To configure a second on-chip memory, in the list of available memory components, double-click **On-Chip Memory (RAM or ROM)**. The On-Chip Memory configuration wizard appears.

8. Configure the settings that are listed in Table 2. Unlike the tightly coupled instruction memory, this memory is single-port. Total memory size for tightly coupled data memory is twice the size of tightly coupled instruction memory at 8 KB.

Table 2. On-Chip Memory Default Settings

Properties		Configuration Settings
Memory type	**RAM**	Turn this option on
	Dual-Port Access	Turn this option off
	Read During Write Mode	This option is **DONT_CARE**
	Block type	**Auto**
Size	**Memory Width**	**32**
	Total memory size	**8 KB**
Read latency	**Slave s1**	**1**
Memory initialization	**Initialize memory content**	Turn this option off
	Enable non-default initialization file	Leave this option off
	Enable In-System Memory Content Editor feature	Leave this option off

9. Click **Finish** to close the On-Chip Memory configuration wizard.
10. Rename the **onchip_memory2_0** component to **tightly_coupled_data_memory**.

Connect and Position the Tightly Coupled Memories

To associate the masters with the tightly coupled memories, perform the following steps:

1. To facilitate creating connections between the tightly coupled memory and the Nios II processor, click each new tightly coupled memory and click **Move Up** to move the individual memories just below the cpu component.
2. If necessary, click the **+** to expand the **tightly_coupled_instruction_memory** component.
3. Using the connections panel in Qsys, connect the `s1` port of **tightly_coupled_instruction_memory** to the **tightly_coupled_instruction_master_0** component listed under the cpu component. To connect a port, click the empty dot at the intersection of the `s1` port and the port you want to connect.
4. Similarly, connect the `s2` port of **tightly_coupled_instruction_memory** to the cpu `data_master` port. This connection is shown in Figure 1 on page 1–3 as the Avalon-MM connection between the Avalon-MM data master port and the Avalon-MM slave port on the tightly coupled instruction memory. Port `s2` of this dual-port memory is an Avalon-MM slave port, not a tightly coupled slave port because `s2` connects to an Avalon-MM master, which is not a tightly coupled master.
5. If necessary, click the **+** to expand the **tightly_coupled_data_memory** component.
6. Connect the `s1` port of **tightly_coupled_data_memory** to **tightly_coupled_data_master_0**.

7. To change the tightly coupled memories to the same clock domain as the cpu, follow these steps:

   a. Click in the **Clock** column next to `clk1` and `clk2` for the `s1` and `s2` ports, respectively, of **tightly_coupled_instruction_memory**. A list of available clock signals appears.

   b. Select the same clock domain as the cpu from the list of available clocks to connect this clock to the slave ports.

   c. Similarly, connect `reset1` and `reset2` for the `s1` and `s2` ports, respectively, to `clk_reset` of the clock source.

   d. Repeat steps a-c for **tightly_coupled_data_memory**.

8. In the **Base** column, enter the base addresses in Table 3 for all tightly coupled memories.

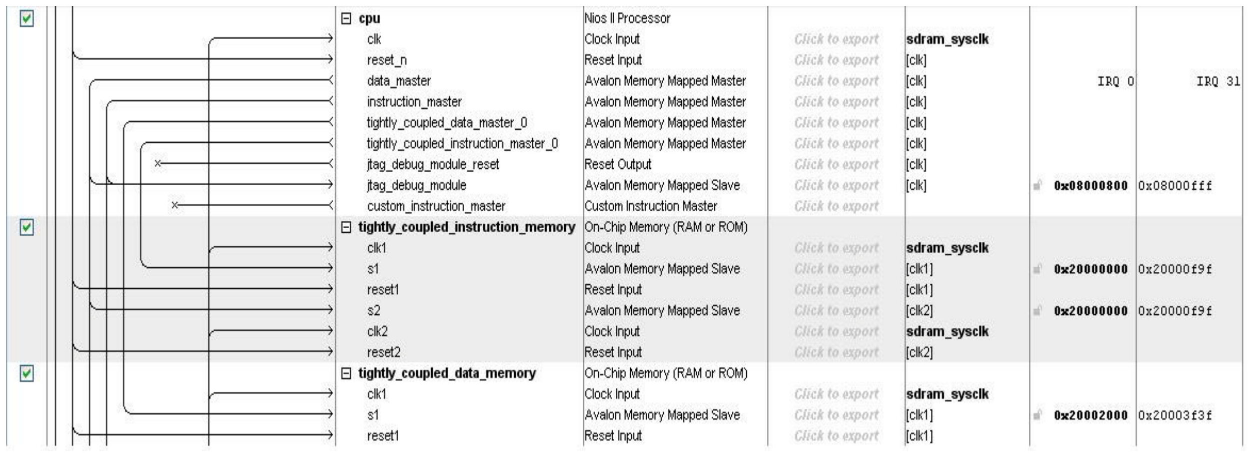
Table 3. Base Addresses for Tightly Coupled Memories

Port	Base
tightly_coupled_instruction_memory `s1`	`0x20000000`
tightly_coupled_instruction_memory `s2`	`0x20000000`
tightly_coupled_data_memory `s2`	`0x20002000`

The **end** addresses automatically update to reflect the memory size that you specified in the configuration wizard. The base address specification is important. Tightly coupled memories must be mapped so that their addresses *do not overlap* with the embedded processor's memories and peripherals that are connected to its Avalon-MM instruction and data masters.

Figure 2 shows the complete system.

Figure 2. Connections for Tightly Coupled Memories



⊟ **cpu**	Nios II Processor				
clk	Clock Input	Click to export	**sdram_sysclk**		
reset_n	Reset Input	Click to export	[clk]		
data_master	Avalon Memory Mapped Master	Click to export	[clk]	IRQ 0	IRQ 31
instruction_master	Avalon Memory Mapped Master	Click to export	[clk]		
tightly_coupled_data_master_0	Avalon Memory Mapped Master	Click to export	[clk]		
tightly_coupled_instruction_master_0	Avalon Memory Mapped Master	Click to export	[clk]		
jtag_debug_module_reset	Reset Output	Click to export	[clk]		
jtag_debug_module	Avalon Memory Mapped Slave	Click to export	[clk]	**0x08000800**	0x08000fff
custom_instruction_master	Custom Instruction Master	Click to export			
⊟ **tightly_coupled_instruction_memory**	On-Chip Memory (RAM or ROM)				
clk1	Clock Input	Click to export	**sdram_sysclk**		
s1	Avalon Memory Mapped Slave	Click to export	[clk1]	**0x20000000**	0x20000f9f
reset1	Reset Input	Click to export	[clk1]		
s2	Avalon Memory Mapped Slave	Click to export	[clk2]	**0x20000000**	0x20000f9f
clk2	Clock Input	Click to export	**sdram_sysclk**		
reset2	Reset Input	Click to export	[clk2]		
⊟ **tightly_coupled_data_memory**	On-Chip Memory (RAM or ROM)				
clk1	Clock Input	Click to export	**sdram_sysclk**		
s1	Avalon Memory Mapped Slave	Click to export	[clk1]	**0x20002000**	0x20003f3f
reset1	Reset Input	Click to export	[clk1]		

To simplify address decoding, map the high-order address bit to a unique location. By limiting the decoding logic to one bit, you minimize the effect of address decoding on f_{MAX}. The Nios II component works correctly even if the address map is not optimal; however, it displays a warning during system generation.

As an example of optimal address mapping, if all the normal memories and peripherals in your system occupy addresses below 0x2000000, mapping your tightly coupled memories at addresses from 0x2000000 and above satisfies this requirement.

9. To set the exception address of the Nios II processor, open the Nios II processor's configuration wizard. On the **Core Nios II** tab, in the **Exception vector memory:** list, select **tightly_coupled_instruction_memory_s1**.

   The address **Offset** fields are populated automatically and are indexed from the base address specified for the memory module on the **System Contents** tab.

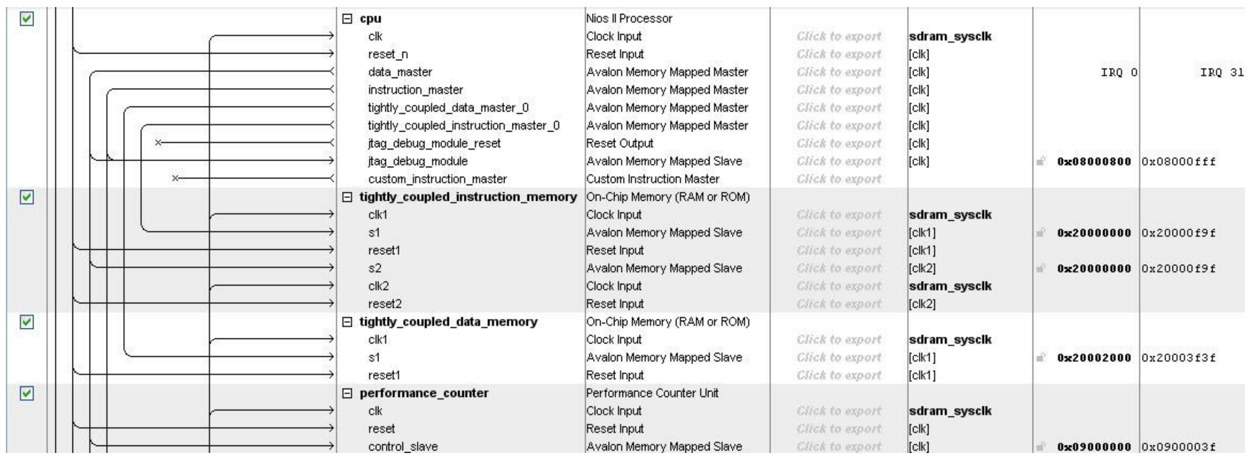
Add a Performance Counter

Add a performance counter peripheral so you can compare the performance of the reads and writes to tightly coupled memory to other memories. To add the performance counter, complete the following steps:

1. In the **Component Library** tab, click **Peripherals** to expand the list of available components.
2. Under **Debug and Performance**, double-click **Performance Counter Unit** to open the Performance Counter configuration wizard.
3. Click **Finish**, accepting the default setting of the **3** simultaneously-measured sections.
4. Be sure that the component is named **performance_counter**.
5. Move the component up just below the tightly coupled memory components. Connect the `control_slave` port of the **performance_counter** to the cpu's `data_master` port. Figure 3 shows these connections.
6. In **performance_counter**, connect `clk` to the same clock domain as cpu and `reset` to `clk_reset` of the clock source.

7. In the **Base** column, change the base address of **performance_counter** to `0x09000000`.

Figure 3. Connections for the Performance Counter



✓	⊟ **cpu**	Nios II Processor				
	clk	Clock Input	Click to export	**sdram_sysclk**		
	reset_n	Reset Input	Click to export	[clk]		
	data_master	Avalon Memory Mapped Master	Click to export	[clk]	IRQ 0	IRQ 31
	instruction_master	Avalon Memory Mapped Master	Click to export	[clk]		
	tightly_coupled_data_master_0	Avalon Memory Mapped Master	Click to export	[clk]		
	tightly_coupled_instruction_master_0	Avalon Memory Mapped Master	Click to export	[clk]		
	jtag_debug_module_reset	Reset Output	Click to export	[clk]		
	jtag_debug_module	Avalon Memory Mapped Slave	Click to export	[clk]	**0x08000800**	0x08000fff
	custom_instruction_master	Custom Instruction Master	Click to export			
✓	⊟ **tightly_coupled_instruction_memory**	On-Chip Memory (RAM or ROM)				
	clk1	Clock Input	Click to export	**sdram_sysclk**		
	s1	Avalon Memory Mapped Slave	Click to export	[clk1]	**0x20000000**	0x20000f9f
	reset1	Reset Input	Click to export	[clk1]		
	s2	Avalon Memory Mapped Slave	Click to export	[clk2]	**0x20000000**	0x20000f9f
	clk2	Clock Input	Click to export	**sdram_sysclk**		
	reset2	Reset Input	Click to export	[clk2]		
✓	⊟ **tightly_coupled_data_memory**	On-Chip Memory (RAM or ROM)				
	clk1	Clock Input	Click to export	**sdram_sysclk**		
	s1	Avalon Memory Mapped Slave	Click to export	[clk1]	**0x20002000**	0x20003f3f
	reset1	Reset Input	Click to export	[clk1]		
✓	⊟ **performance_counter**	Performance Counter Unit				
	clk	Clock Input	Click to export	**sdram_sysclk**		
	reset	Reset Input	Click to export	[clk]		
	control_slave	Avalon Memory Mapped Slave	Click to export	[clk]	**0x09000000**	0x0900003f

Generate the Qsys System

To generate and compile the hardware system, perform the following steps:

1. In the **Generation** tab, click **Generate**. Click **Save** to save the changes in the Qsys system.
2. In the Quartus II software, on the **Processing** menu, click **Start Compilation**.
3. When the Quartus II software compilation is finished, on the Tools menu, click **Programmer** to program the newly generated **niosii_ethernet_standard_**<*board*>**.sof** into the FPGA.

Run the Tightly Coupled Memories Examples from the Nios II Command Shell

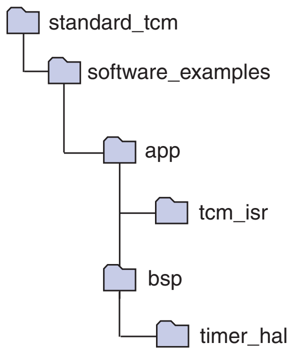
To run the tightly coupled memories example from the Nios II command shell, perform the following steps:

1. To open a Nios II command shell under Windows, in the Start menu, point to **Programs** > **Altera** > **Nios II EDS** <*version number*>, and click **Nios II** <*version number*> **Command Shell**.
2. Navigate to the working directory for your project. The following steps refer to this directory as <*project_directory*>.
3. Ensure that the working directory and all subdirectories are writable by typing the following command:

```
chmod -R +w . ↵
```

4. Download the **tcm.zip** file from the Altera web page for the Tightly Coupled Memory tutorial and unzip it into the <*project_directory*> directory. Figure 4 shows the directory structure for the unzipped files.

Figure 4. Project Directory after Unzipping the Files



5. Change to the **software_examples/app/tcm_isr** subdirectory of your <*project_directory*> by typing the following command:

```
cd software_examples/app/tcm_isr ↵
```

6. Create and build the application by typing the following command:

```
./create-this-app ↵
```

7. The linker script file, **linker.x** in the **bsp/timer_hal** directory includes a new `isrs_region` located in tightly coupled instruction memory which is adjacent to the `tightly_coupled_instruction_memory` region. Example 1 shows the new region.

Example 1. isrs_region Listing in linker.x File

```
MEMORY
{
:
:
tightly_coupled_instruction_memory : ORIGIN = 0x20000120, LENGTH = 1664
timer_isrs_region : ORIGIN = 0x200007a0, LENGTH = 2048
:
:
}
```

8. The **tcmisr.objdump** in the **app/tcm_isr** directory defines the `.isrs` section located in the tightly coupled instruction memory. Example 2 shows part of this file.

Example 2. isrs Section Listing in tcm_isr.objdump File

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .entry        00000000  00000000  00000000  00001000  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .exceptions   000001a0  20000120  20000120  00013120  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .isrs         000000c4  200007a0  200007a0  000137a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .text         0000fea4  10000000  10000000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .rodata       00000650  1000fea4  1000fea4  00010ea4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .rwdata       00001ba0  100104f4  100104f4  000114f4  2**2
                  CONTENTS, ALLOC, LOAD, DATA, SMALL_DATA
  6 .bss          00000154  10012094  10012094  00013094  2**2
                  ALLOC, SMALL_DATA
```

Program and Run the Tightly Coupled Memory Project

To program and run a tightly coupled memory project, perform the following steps:

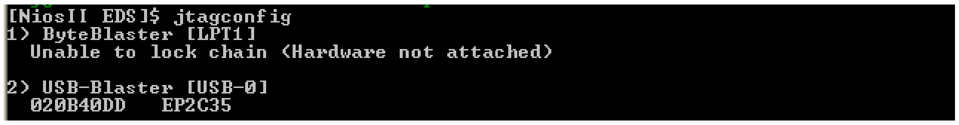
1. To open a terminal session to capture messages from the Nios II processor, type the following command:

   `nios2-terminal ↵`

   If your development board includes more than one JTAG cable, you must specify which cable you are communicating with as an argument to the `nios2-terminal` command. To do so, first type the following command:

   `jtagconfig -r ↵`

Figure 5. jtagconfig Output



```
[NiosII EDS]$ jtagconfig
1) ByteBlaster [LPT1]
  Unable to lock chain (Hardware not attached)

2) USB-Blaster [USB-0]
  020B40DD   EP2C35
```

   Figure 5 shows some sample output from the `jtagconfig` command. This output shows that the active JTAG cable is number 2. Substitute the number of your JTAG cable for the <*cable_number*> variable in the following command:

   `nios2-terminal -c <cable_number> ↵`

2. In your first shell, enter the following command if you have a single JTAG cable:

   `nios2-download -g tcm_isr.elf ↵`

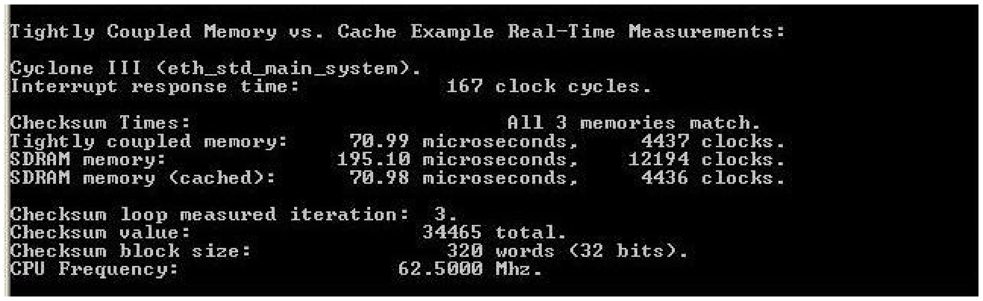
   For development boards with more than one JTAG cable, enter the following command:

   `nios2-download -c <cable_number> -g tcm_isr.elf ↵`

Figure 6 is a printout of the statistics that shows the higher speeds obtained by leveraging tightly coupled memories. Note that the number of clock cycles for tightly coupled memory is very similar to that of the cached memory. The result demonstrates that tightly coupled memories allow fixed low-latency read access to executable code as well as fixed low-latency read, write, or read and write access to data.

The timing numbers output varies between Nios development boards.

Figure 6. Tightly Coupled Memory versus Cache Example Real-Time Measures



```
Tightly Coupled Memory vs. Cache Example Real-Time Measurements:

Cyclone III (eth_std_main_system).
Interrupt response time:            167 clock cycles.

Checksum Times:                          All 3 memories match.
Tightly coupled memory:     70.99 microseconds,     4437 clocks.
SDRAM memory:              195.10 microseconds,    12194 clocks.
SDRAM memory (cached):      70.98 microseconds,     4436 clocks.

Checksum loop measured iteration:  3.
Checksum value:                  34465 total.
Checksum block size:               320 words (32 bits).
CPU Frequency:                 62.5000 Mhz.
```

Understanding the Tcl Scripts

The following sections describe creating special memory regions for the timer memory interrupt service routines and timer definitions and locating the exception stack in the tightly coupled memory.

Timer Memory

The **timer_memory_section.tcl** script is located in the **bsp/timer_hal** directory. This Tcl script reserves 2048 bytes of the tightly coupled instruction memory. The reserved space is used to store the timer interrupt service routines.

The **timer_memory_section.tcl** script takes the `tightly_coupled_instruction` memory region and separates out 2048 bytes of the memory region into a new region called `timer_isr_region`. The next line of code adds a section mapping the `.isrs` section to the `timer_isr_region`. Example 3 shows this code.

Example 3. Timer ISR Region

```
# Create tightly_coupled_memory region.
add_memory_region tightly_coupled_instruction_memory $slave $offset $new_span
# Create a second region called timer_isr memory_region.
add_memory_region timer_isrs_region $slave $split_offset $split_span
# Create memory mapping to map .isrs to timer_isrs_region.
add_section_mapping .isrs timer_isrs_region
```

The **timer_interrupt_latency.h** file is also updated to reflect the change in the section mapping of the `timer_interrupt_latency_irq()` timer interrupt service routine to `.isrs` instead of `.exceptions`. The timer interrupt service routines are now stored in the `timer_isr_region`.

The interrupt service routines must be located in the new `.isrs` section. Otherwise, the linker uses the default setting, defeating the purpose of declaring a special memory section for the interrupt service routine.

To locate the interrupt service routines in the new `.isrs` section, complete the following steps:

1. Add a section mapping to map the `.isrs` section to the newly added memory region.
2. Edit your source files to ensure the interrupt service routines are mapped to the new memory section.
3. Compile your project files and check the **linker.x** and **tcm_isr.objdump** files to ensure that the section mapping and memory regions are declared correctly and contain the interrupt service routine.

For more information about linker memory regions, refer to the *Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*.

Exception Stack

The **timer_memory_section.tcl** script locates the exception stack in the `tightly_coupled_data_memory` region. Example 4 shows this code.

Example 4. Exception Stack

```
# Locate the exception stack to tightly coupled data memory.
set_setting hal.linker.enable_exception_stack TRUE
set_setting hal.linker.exception_stack_memory_region_name tightly_coupled_data_memory.
aset_setting hal.linker.exception_stack_size 1024
```

Timer Definitions

The following sections describe `peripheral_subsystem_sys_clk_timer` and `peripheral_subsystem_high_res_timer`.

peripheral_subsystem_sys_clk_timer

The **timer_definition.tcl** script is located in the **bsp/timer_hal** directory. The script defines the timers as follows:

```
set_setting hal.sys_clk_timer peripheral_subsystem_sys_clk_timer
set_setting hal.timestamp_timer none
```

This script is essential for the clocks definitions. The software driver `hal.sys_clk_timer` must be driven by the hardware clock named `peripheral_subsystem_sys_clk_timer`. Connecting `hal.sys_clk_timer` to any other hardware timer results in a compilation error. The following exercise demonstrates this point.

1. Delete or rename the **Makefile** in the **app/tcm_isr** folder. Then delete or rename **public.mk** from the **bsp/timer_hal** folder.

2. Open the **timer_definition.tcl** file and change `peripheral_subsystem_sys_clk_timer` to `peripheral_subsystem_high_res_timer` as follows:

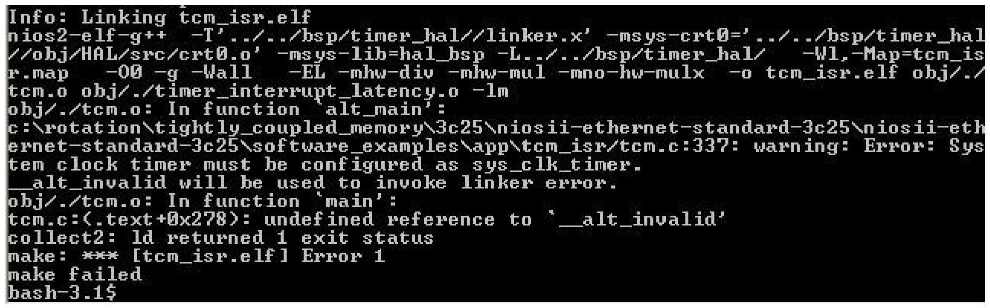
```
set_setting hal.sys_clk_timer peripheral_subsystem_high_res_timer
set_setting hal.timestamp_timer none
```

3. Save **timer_definition.tcl**.
4. Return to your shell and recreate the application by typing:

```
./create-this-app ↵
```

5. Figure 7 shows the error. Setting `hal.sys_clk_timer` to any other timers except for `peripheral_subsystem_sys_clk_timer` results in the same error message.

Figure 7. Error Message after Changing the hal.sys_clk_timer



```
Info: Linking tcm_isr.elf
nios2-elf-g++  -T'../../bsp/timer_hal//linker.x' -msys-crt0='../../bsp/timer_hal
//obj/HAL/src/crt0.o' -msys-lib=hal_bsp -L../../bsp/timer_hal/   -Wl,-Map=tcm_is
r.map   -O0 -g -Wall   -EL -mhw-div -mhw-mul -mno-hw-mulx  -o tcm_isr.elf obj/./
tcm.o obj/./timer_interrupt_latency.o -lm
obj/./tcm.o: In function `alt_main':
c:\rotation\tightly_coupled_memory\3c25\niosii-ethernet-standard-3c25\niosii-eth
ernet-standard-3c25\software_examples\app\tcm_isr/tcm.c:337: warning: Error: Sys
tem clock timer must be configured as sys_clk_timer.
__alt_invalid will be used to invoke linker error.
obj/./tcm.o: In function `main':
tcm.c:(.text+0x278): undefined reference to `__alt_invalid'
collect2: ld returned 1 exit status
make: *** [tcm_isr.elf] Error 1
make failed
bash-3.1$
```

peripheral_subsystem_high_res_timer

The hardware timer called `peripheral_subsystem_high_res_timer` calculates interrupt latency. `timer_interrupt_latency_init()`, defined in **timer_interrupt_latency.c**, installs an interrupt service routine to handle `peripheral_subsystem_high_res_timer`. Therefore, `peripheral_subsystem_high_res_timer` must not be tied to the software timestamp driver, `hal.timestamp_timer`; it is set to **none**. Because `peripheral_subsystem_sys_clk_timer` is used for `hal.sys_clk_timer`, it must not be used for `hal.timestamp_timer`.

The following exercise shows this point:

1. If you have not already done so, delete or rename **Makefile** in the **app/tcm_isr** folder. Delete or rename **public.mk** in the **bsp/timer_hal** folder.
2. Open the **timer_definition.tcl** file and change the setting of `hal.timestamp_timer` from `none` to `peripheral_subsystem_high_res_timer` as follows:

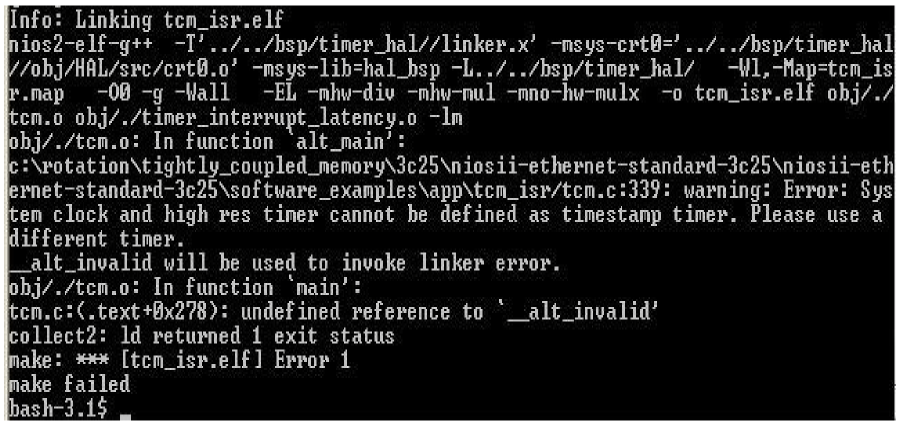
```
set_setting hal.sys_clk_timer peripheral_subsystem_sys_clk_timer
set_setting hal.timestamp_timer peripheral_subsystem_high_res_timer
```

3. Save **timer_definition.tcl**.
4. Return to your shell and recreate the application by typing the following command:

```
./create-this-app ↵
```

5. Figure 8 shows the error that you get. Setting `hal.timestamp_timer` to `peripheral_subsystem_sys_clk_timer` or `peripheral_subsystem_high_res_timer` results in the same error message. Setting `hal.timestamp_timer` to other hardware timers in the system will not result in the error message in Figure 8.

Figure 8. Error Message after Changing the hal.timestamp_timer



```
Info: Linking tcm_isr.elf
nios2-elf-g++  -T'../../bsp/timer_hal//linker.x' -msys-crt0='../../bsp/timer_hal
//obj/HAL/src/crt0.o' -msys-lib=hal_bsp -L../../bsp/timer_hal/   -Wl,-Map=tcm_is
r.map   -O0 -g -Wall   -EL -mhw-div -mhw-mul -mno-hw-mulx  -o tcm_isr.elf obj/./
tcm.o obj/./timer_interrupt_latency.o -lm
obj/./tcm.o: In function `alt_main':
c:\rotation\tightly_coupled_memory\3c25\niosii-ethernet-standard-3c25\niosii-eth
ernet-standard-3c25\software_examples\app\tcm_isr/tcm.c:339: warning: Error: Sys
tem clock and high res timer cannot be defined as timestamp timer. Please use a
different timer.
__alt_invalid will be used to invoke linker error.
obj/./tcm.o: In function `main':
tcm.c:(.text+0x278): undefined reference to `__alt_invalid'
collect2: ld returned 1 exit status
make: *** [tcm_isr.elf] Error 1
make failed
bash-3.1$ _
```

This chapter provides additional information about the document and Altera.

Document Revision History

The following table shows the revision history for this tutorial.

Date	Version	Changes
July 2011	2.0	■ Converted to new 8.5 x 11" template. ■ Replaced references to SOPC Builder with Qsys. ■ Minor text edits.
April 2009	1.1	Minor updates to reflect latest software updates.
July 2008	1.0	Updated to use Quartus II 8.0 and SOPC Builder 8.0. Revised design example instructions to the command line tools instead of the Nios II IDE.
July 2005		Initial release.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact *(1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.

Visual Cue	Meaning
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, <*file name*> and <*project name*>**.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
☞	The hand points to information that requires special attention.
?	A question mark directs you to a software help system with related information.
👣	The feet direct you to another document or website with related information.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚡ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
✉	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.