Using Tightly Coupled Memory with the Nios II Processor Tutorial

TU-N2060305-1.1

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

The Nios II architecture provides tightly coupled master ports that deliver guaranteed, fixed, low-latency access to on-chip memory for performance critical applications. Tightly coupled masters can connect to instruction memory and data memory allowing fixed, low-latency read access to executable code as well as fixed low-latency read, write, or read and write access to data. Tightly coupled masters are additional instruction or data master ports on the Nios II core, separate from the embedded processor's instruction and data master ports.

This document assumes you are familiar with the Nios II tightly coupled memory. For details, see the *Processor Architecture* chapter in the *Nios II Processor Reference Handbook*.

Reasons for Using Tightly Coupled Memory

You can implement a wide variety of functions or modules using tightly coupled memories. If you use tightly coupled memory to implement the following hardware features, you 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 loops
- Constant access time; 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

Cache memory provides a generalized speed enhancement for all code. On the other hand, tightly coupled memory uses a dedicated memory block, guaranteeing a particular section of code or data has fast access times and achieves high performance. Locating 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 provides guidelines and limitations which you need to understand when you design hardware and software with tightly coupled memory.

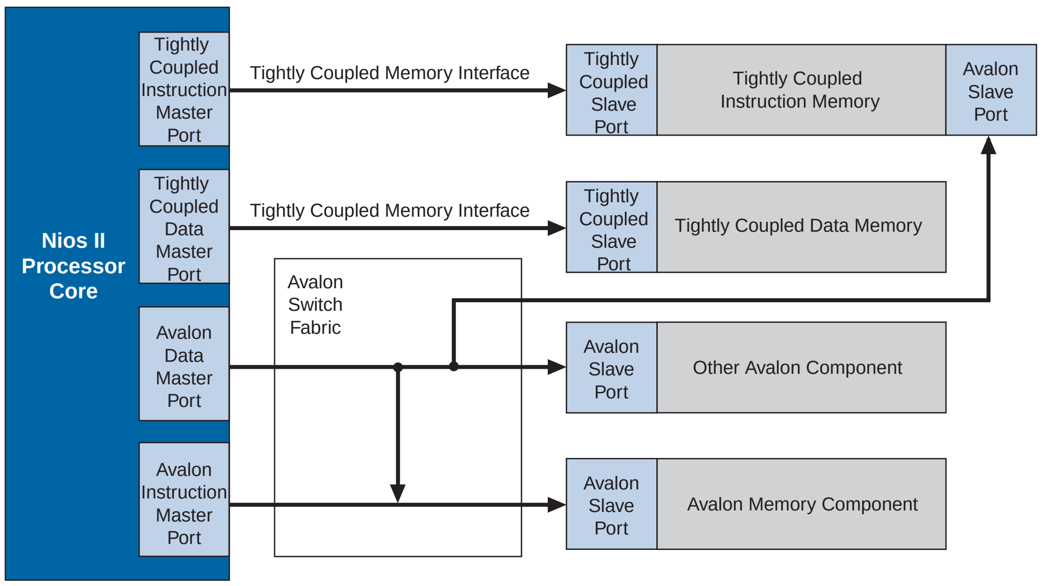
Hardware Guidelines

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 CPU.
- An on-chip memory SOPC Builder 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 at most one tightly coupled master even if it is a dual port memory.
- Whether data or instruction tightly coupled masters are available depends on the type of Nios II core.
- When using the **On-Chip Memory** component as a tightly coupled memory for Nios II, you must always configure it as a RAM, and not a ROM. Tightly coupled memories configured as ROM fail.
- To conserve logic elements, it is better to have one 2 KByte tightly coupled memory, than 2 tightly coupled memories of size 1 KByte.

Figure 1 is a block diagram of a simple Nios II system, which includes tightly coupled memories and other SOPC Builder 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:

```
extern void timer_interrupt_latency_irq (void* base, alt_u32 id)
attribute ((section (".exceptions")));
```

SOPC Builder 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.

SOPC Builder creates additional linker sections with address mappings that are controlled by SOPC Builder. For the case of the `.exceptions` section, the physical address offset and memory module in which to base that linker section is manipulated through SOPC Builder. You locate `.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 additional details on the C section attribute, see 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.

Restrictions

You must observe a 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 1 cycle, a write latency of 0 cycles, and no wait states.

When tightly coupled memory is present, the Nios II core decodes addresses internally to determine if 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 Figure 1 illustrates. 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 Embedded Design Suite (EDS) which makes development and debugging difficult. Without a second port on the tightly coupled instruction memory, no data master has access to the memory, meaning the user has no way of viewing 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 SOPC Builder that uses two tightly coupled memories, one instruction and one data. These two tightly coupled memories are connected to the Nios II processor as shown in Figure 1. 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.

In this section you perform the following steps:

1. Modify an existing reference design to include tightly coupled memories.
2. Create the tightly coupled memories in SOPC Builder.
3. Connect the tightly coupled memories to 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 your hardware design.
10. Change the Tcl scripts and recompile to demonstrate how the timer settings work.

Hardware and Software Requirements

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

- Nios II EDS version 8.0 or later
- Quartus® II software version 8.0 or later
- Any Nios II development board

Modify the Example Design to Include Tightly Coupled Memories

First, you create a new hardware reference design with tightly coupled memories that is based on any of the **standard** reference designs installed with the Nios II EDS. To create this modified reference design, perform the following steps:

1. In your host computer file system, locate the **standard** design directory for your chosen development board and HDL. For example, on a Windows PC **C:\altera\**<*version*>**\nios2eds\examples\verilog\niosII_cycloneII_2c35\standard** contains the Verilog HDL design files for the Nios Development Board, Cyclone II Edition.
2. Copy all the files from the **standard** directory to a new directory named **standard_tcm**.
3. Choose **Programs > Altera > Quartus II** <*version*> (Windows Start menu) to run the Quartus II software. You can also use the Quartus II Web Edition software.
4. On the File menu, click **Open Project** and browse to the **standard_tcm\**<*board_name*>**.qpf** project file.
5. On the Tools menu, click **SOPC Builder**. The SOPC Builder standard design appears in SOPC Builder.
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. On the **Core Nios II** tab, select **Nios II/f**.
8. Click the **Caches and Memory Interfaces** tab.
9. Turn on **Include tightly coupled instruction master port(s)**.
10. Turn on **Include tightly coupled data master port(s)**.
11. 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 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 **System Contents** 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.

Table 1. On-Chip Memory Default Settings

Properties		Configuration Settings
Memory type	**ROM**	Select this option
	Dual-port access	Turn this option on
	Read During Write Mode	Select **DONT_CARE**
	Block type	Select **Auto**
	Initialize memory content	Turn this option on
Size	**Data width**	Select **32**
	Total memory size	Specify **4 KBytes**
Read latency	**Slave s1**	Select **1**
	Slave s2	Select **1**
Memory initialization	**Enable non-default initialization file**	Leave this option turned off
	Enable In-System Memory Content Editor feature	Leave this option turned off

3. Click **Finish** to close the configuration wizard.
4. On the **System Contents** tab, right-click the **onchip_mem** component.

   In this tutorial, you *must* name the components exactly. If your component names differ from the names printed here, the software example will not work.

5. Right-click **onchip_mem** and rename the component to **tightly_coupled_instruction_memory**.
6. To configure a second on-chip memory, in the list of available memory components, double-click **On-Chip Memory**. The On-Chip Memory configuration wizard appears.
7. Specify the settings 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 KBytes.

Table 2. On-Chip Memory Default Settings

Properties		Configuration Settings
Memory type	**RAM**	Select this option
	Dual-port access	Turn this option off
	Read During Write Mode	Select **DONT_CARE**
	Block type	Select **Auto**
	Initialize memory content	Turn this option off
Size	**Data width**	Specify **32**
	Total memory size	Specify **8 KBytes**
Read latency	**Slave s1**	Select **1**
	Slave s2	Select **1**

Table 2. On-Chip Memory Default Settings

Properties		Configuration Settings
Memory initialization	**Enable non-default initialization file**	Leave this option turned off
	Enable In-System Memory Content Editor feature	Leave this option turned off

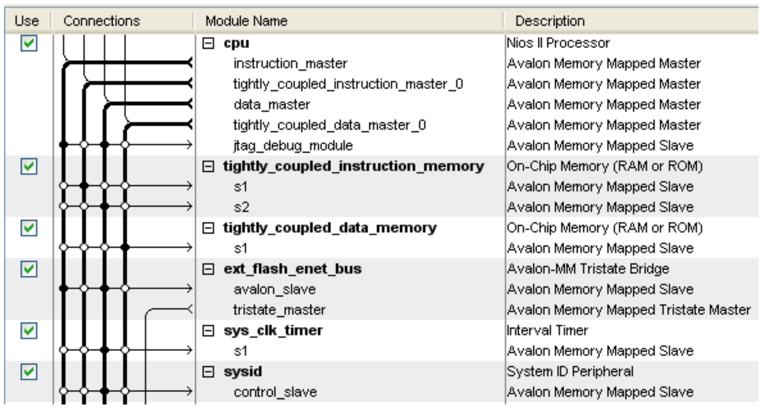
8. Click **Finish** to close the On-Chip Memory configuration wizard.
9. Rename the **onchip_mem** component to **tightly_coupled_data_memory**.

Connect and Position the Tightly Coupled Memories

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

1. To simplify 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 **+** to expand the tightly_coupled_instruction_memory component.
3. Using the patch-panel connection matrix in SOPC Builder, disconnect the `s1` port of **tightly_coupled_instruction_memory** from all masters except for the **tightly_coupled_instruction_master_0** component listed under the **cpu** component. To disconnect a port, click on the filled dot at the intersection of the `s1` port and the port you need to disconnect. The remaining connection is the tightly coupled memory interface shown in Figure 1 for the tightly coupled instruction memory, connecting the tightly coupled instruction master port to the tightly coupled slave port on the tightly coupled instruction memory. Figure 2 illustrates the connections for the `s1` port.

Figure 2. Connections for the s1 Port



Use	Connections	Module Name	Description
✓		**cpu**	Nios II Processor
		instruction_master	Avalon Memory Mapped Master
		tightly_coupled_instruction_master_0	Avalon Memory Mapped Master
		data_master	Avalon Memory Mapped Master
		tightly_coupled_data_master_0	Avalon Memory Mapped Master
		jtag_debug_module	Avalon Memory Mapped Slave
✓		**tightly_coupled_instruction_memory**	On-Chip Memory (RAM or ROM)
		s1	Avalon Memory Mapped Slave
		s2	Avalon Memory Mapped Slave
✓		**tightly_coupled_data_memory**	On-Chip Memory (RAM or ROM)
		s1	Avalon Memory Mapped Slave
✓		**ext_flash_enet_bus**	Avalon-MM Tristate Bridge
		avalon_slave	Avalon Memory Mapped Slave
		tristate_master	Avalon Memory Mapped Tristate Master
✓		**sys_clk_timer**	Interval Timer
		s1	Avalon Memory Mapped Slave
✓		**sysid**	System ID Peripheral
		control_slave	Avalon Memory Mapped Slave

4. Similarly, disconnect the `s2` port of the **tightly_coupled_instruction_memory** from all masters except the cpu `data_master` port. This connection is shown in Figure 1 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, rather than a tightly coupled master.

5. Click **+** to expand the **tightly_coupled_data_memory** component.
6. Disconnect the `s1` port of **tightly_coupled_data_memory** from all masters except **tightly_coupled_data_master_0**. The remaining connection is the tightly coupled memory interface shown in Figure 1 for the tightly coupled data memory, connecting the tightly coupled `data_master` port to the tightly coupled `s1` port on the tightly coupled data memory.
7. To change the tightly coupled memories to the same clock domain as the cpu, complete following these steps:
   a. Click in the **Clock** column next to the `s1` and `s2` ports. A list of available clock signals appears.
   b. Select **pll_c0** from the list of available clocks to connect this clock the slave ports.
8. In the **Base** column, enter the base addresses in Table 3 for all tightly coupled memories:

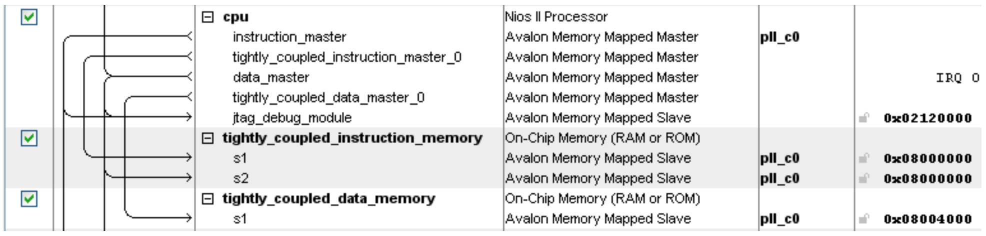
Table 3. Base Addresses for Tightly Coupled Memories

Port	Base
tightly_coupled_instruction_memory `s1`	`0x08000000`
tightly_coupled_instruction_memory `s2`	`0x08000000`
tightly_coupled_data_memory `s2`	`0x08004000`

The **end** addresses automatically update to reflect the memory size that you specify 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 4 on page 9 illustrates the complete system.

Figure 3. Connections for tightly coupled memories



✓	cpu	Nios II Processor		
	instruction_master	Avalon Memory Mapped Master	pll_c0	
	tightly_coupled_instruction_master_0	Avalon Memory Mapped Master		
	data_master	Avalon Memory Mapped Master		IRQ 0
	tightly_coupled_data_master_0	Avalon Memory Mapped Master		
	jtag_debug_module	Avalon Memory Mapped Slave		0x02120000
✓	tightly_coupled_instruction_memory	On-Chip Memory (RAM or ROM)		
	s1	Avalon Memory Mapped Slave	pll_c0	0x08000000
	s2	Avalon Memory Mapped Slave	pll_c0	0x08000000
✓	tightly_coupled_data_memory	On-Chip Memory (RAM or ROM)		
	s1	Avalon Memory Mapped Slave	pll_c0	0x08004000

To simplify address decoding, you can 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 processor works correctly even if the address map is not optimal; however, a warning is displayed 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.

1. To set the exception address, on the **Core Nios II** tab, in the **Exception Vector: Memory**: list, select **tightly_coupled_instruction_memory_s1**.

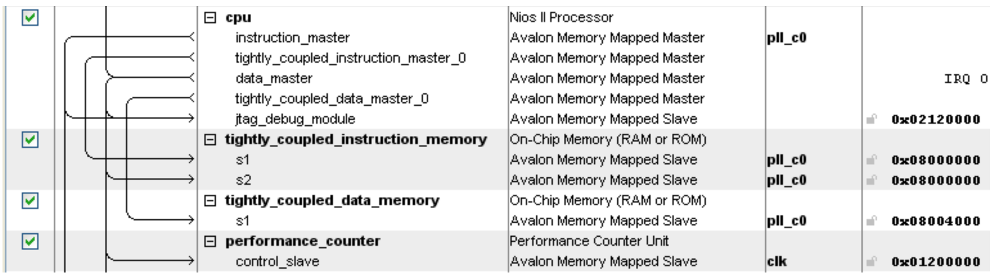
2. Note that the address **Offset** fields are specified automatically and are indexed from the base address specified for the memory module on the **System Contents** tab.

Add a Performance Counter

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

1. Click the **System Contents** tab.
2. In the list of available components, click **Peripherals** to expand the list of available components.
3. Under **Debug and Performance**, double-click **Performance Counter Unit** to open the Performance Counter configuration wizard.
4. Click **Finish**, accepting the default setting of the **3** simultaneously-measured sections.
5. Make sure that the component is named **performance_counter.**
6. Use the **Move Up** button to move the component up just below the tightly coupled memory components. The `control_slave` port of the performance_counter is automatically connected to the tightly_coupled_instruction_memory `s2` port, and the cpu's `data_master` and `jtag_debug_module` ports. Figure 4 illustrates these connections.
7. Connect the `control_slave` port to the **pll_c0** clock.

Figure 4. Connections for the Performance Counter



✓	**cpu**	Nios II Processor		
	instruction_master	Avalon Memory Mapped Master	**pll_c0**	
	tightly_coupled_instruction_master_0	Avalon Memory Mapped Master		
	data_master	Avalon Memory Mapped Master		IRQ 0
	tightly_coupled_data_master_0	Avalon Memory Mapped Master		
	jtag_debug_module	Avalon Memory Mapped Slave		**0x02120000**
✓	**tightly_coupled_instruction_memory**	On-Chip Memory (RAM or ROM)		
	s1	Avalon Memory Mapped Slave	**pll_c0**	**0x08000000**
	s2	Avalon Memory Mapped Slave	**pll_c0**	**0x08000000**
✓	**tightly_coupled_data_memory**	On-Chip Memory (RAM or ROM)		
	s1	Avalon Memory Mapped Slave	**pll_c0**	**0x08004000**
✓	**performance_counter**	Performance Counter Unit		
	control_slave	Avalon Memory Mapped Slave	**clk**	**0x01200000**

Generate the SOPC Builder System

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

1. In SOPC Builder, click **Next**.
2. On the **System Generation** tab, click **Generate**. After SOPC Builder reports successful system generation, **save** the system.
3. In the Quartus II software, on the Processing menu, click **Start Compilation.**

4. When the Quartus II compilation is complete, on the Tools menu click **Programmer** to open the Programmer that allows you to program the newly generated **standard.sof** into an FPGA.

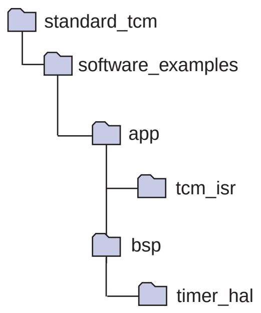
Run the Tightly Couple Memories Examples from Nios II Command Shell

1. To open a Nios II command shell under Windows, in the Start menu, point to **Programs > Altera > Nios II EDS** <*version*>, and click **Nios II Command Shell** <*version*>.
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 and unzip the **tcm.zip** file into the <*project_directory*> directory. A hyperlink to the **tcm.zip** appears next to this application note on the Application Notes web page. Figure 5 illustrates the directory structure for the unzipped files.

Figure 5. 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 = 0x8000020, LENGTH = 2016
timer_isrs_region : ORIGIN = 0x8000800, LENGTH = 2048
.
.
}
```

8. The **tcm_isr.objdump** file in the **app/tcm_isr** directory defines the `.isrs` section located in the tightly coupled instruction memory. Example 2 shows an excerpt from 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 000000d4 2**5
CONTENTS, ALLOC, LOAD, READONLY, CODE
1 .exceptions 000001a8 08000020 08000020 000128c8 2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
2 .text 00010110 04000000 04000000 000000d4 2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
3 .rodata 00000934 04010110 04010110 000101e4 2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .rwdata 00001db0 04010a44 04010a44 00010b18 2**2
CONTENTS, ALLOC, LOAD, DATA, SMALL_DATA
5 .bss 00000108 040127f4 040127f4 000128c8 2**2
ALLOC, SMALL_DATA
6 .isrs 000000c0 08000800 08000800 00012a70 2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Program and Run TCM Project

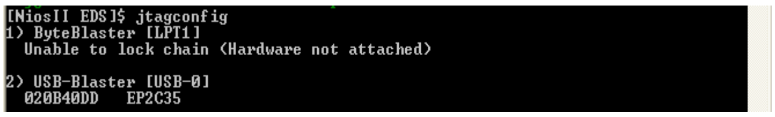
1. You need a second command shell to capture messages from the Nios II processor. (**Programs > Altera > Nios II EDS** <*version*>, then click **Nios II Command Shell** <*version*>). Type the following command:

   `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, type the following commands:

   a. `jtagconf ↵`

Figure 6. jtagconfig Output



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

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

Figure 6 gives sample output from the `jtagconfig` command. This output shows that the active JTAG cable is number 2. Substitute the number of your JTAG for the <*cable_number*> variable in the following command:

b. `nios2-terminal -c` <*cable_number*> ↵

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

   `nios2-download -g tcm.elf` ↵

   or

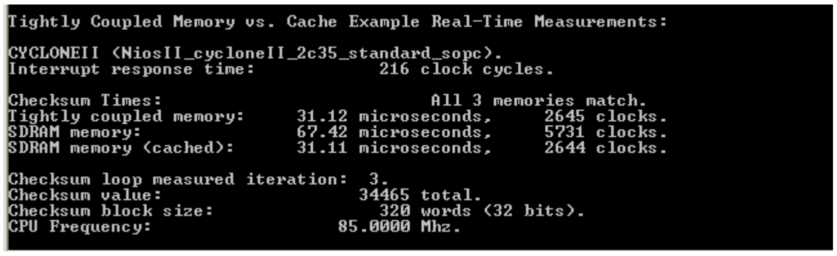
   `nios2-download -c <cable_number> -g tcm.elf` ↵

   for development boards with more than one JTAG cable.

Figure 7 is a printout of the statistics that illustrate 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 7. Tightly Coupled Memory versus Cache Example Real-Time Measures



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

CYCLONEII (NiosII_cycloneII_2c35_standard_sopc).
Interrupt response time:            216 clock cycles.

Checksum Times:                            All 3 memories match.
Tightly coupled memory:     31.12 microseconds,     2645 clocks.
SDRAM memory:               67.42 microseconds,     5731 clocks.
SDRAM memory (cached):      31.11 microseconds,     2644 clocks.

Checksum loop measured iteration:  3.
Checksum value:                   34465 total.
Checksum block size:                320 words (32 bits).
CPU Frequency:                  85.0000 Mhz.
```

Understanding the Tcl Scripts

The following sections discuss creating special memory regions for the timer memory interrupt service routines and timer definitions.

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` defined above. Example 3 shows this code.

Example 3. Timer ISR Region

```
# Create tightly_couple_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_isr_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 `timer_interrupt_latency_irq()` timer interrupt service routine to `.isrs` instead of `.exceptions`. The timer interrupt service routines are now be 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 which defeats 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 make sure that the `isrs` are mapped to the new memory section.
3. Compile your project files by typing the following command in the **bsp/timer_hal** directory:

   `./create-this-bsp` ↵
4. Check the **bsp/timer_hal/linker.x** and **app/tcm_isr/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 *Section 1: Nios II Software Development* in the *Nios II Software Developer's Handbook*.

Timer Definitions

The following sections discuss the `sys_clk_timer` and `high_res_timer`.

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 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 `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 the `sys_clk_timer` to the `high_res_timer` as follows:

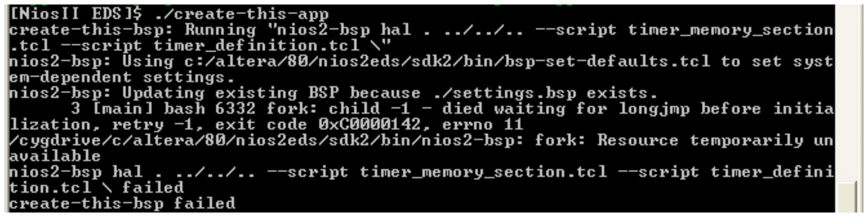
```
set_setting hal.sys_clk_timer high_res_timer
set_setting hal.timestamp_timer none
```

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

```
./create-this-app ↵
```

5. Figure 8 illustrates the error that you see. Setting the `hal.sys_clk_timer` to any other timers except for `sys_clk_timer` results in the same error message.

Figure 8. Error Message after Changing the sys_clk_timer



```
[NiosII EDS]$ ./create-this-app
create-this-bsp: Running "nios2-bsp hal . ../../.. --script timer_memory_section
.tcl --script timer_definition.tcl \"
nios2-bsp: Using c:/altera/80/nios2eds/sdk2/bin/bsp-set-defaults.tcl to set syst
em-dependent settings.
nios2-bsp: Updating existing BSP because ./settings.bsp exists.
      3 [main] bash 6332 fork: child -1 - died waiting for longjmp before initia
lization, retry -1, exit code 0xC0000142, errno 11
/cygdrive/c/altera/80/nios2eds/sdk2/bin/nios2-bsp: fork: Resource temporarily un
available
nios2-bsp hal . ../../.. --script timer_memory_section.tcl --script timer_defini
tion.tcl \ failed
create-this-bsp failed
```

high_res_timer

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

The following exercise illustrates this point:

1. If you have not already done so, delete or rename the **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 `high_res_timer` as follows:

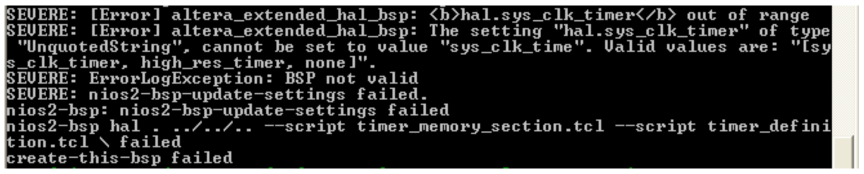
```
set_setting hal.sys_clk_timer sys_clk_timer
set_setting hal.timestamp_timer high_res_timer
```

3. Save **timer_definition.tcl**.
4. Change to the **app/tcm_isr** and recreate the application by typing the following command:

```
./create-this-app ↵
```

Figure 9 illustrates the error that you see. Setting the `hal.timestamp_timer` to `sys_clk_timer` or `high_res_timer` results in the same error message. Setting `hal.timestamp_timer` to other hardware timers in the system prevents the error message below.

Figure 9. Error Message after Changing the timestamp_timer



```
SEVERE: [Error] altera_extended_hal_bsp: <b>hal.sys_clk_timer</b> out of range
SEVERE: [Error] altera_extended_hal_bsp: The setting "hal.sys_clk_timer" of type
 "UnquotedString", cannot be set to value "sys_clk_time". Valid values are: "[sy
s_clk_timer, high_res_timer, none]".
SEVERE: ErrorLogException: BSP not valid
SEVERE: nios2-bsp-update-settings failed.
nios2-bsp: nios2-bsp-update-settings failed
nios2-bsp hal . ../../.. --script timer_memory_section.tcl --script timer_defini
tion.tcl \ failed
create-this-bsp failed
```

References

This application note references the following documents:

Nios II Processor Reference Handbook

Nios II Software Developer's Handbook

Document Revision History

Table 4 shows the revision history for this application note.

Table 4. Document Revision History

Date	Changes Made	Summary of Changes
July 2008, v1.1	Updated to use Quartus II 8.0 and SOPC Builder 8.0. Revised design example instructions to the Software Build Tools instead of the Nios II IDE.	Updated to reflect current state of tools.
July 2005, v1.0	Initial release	—