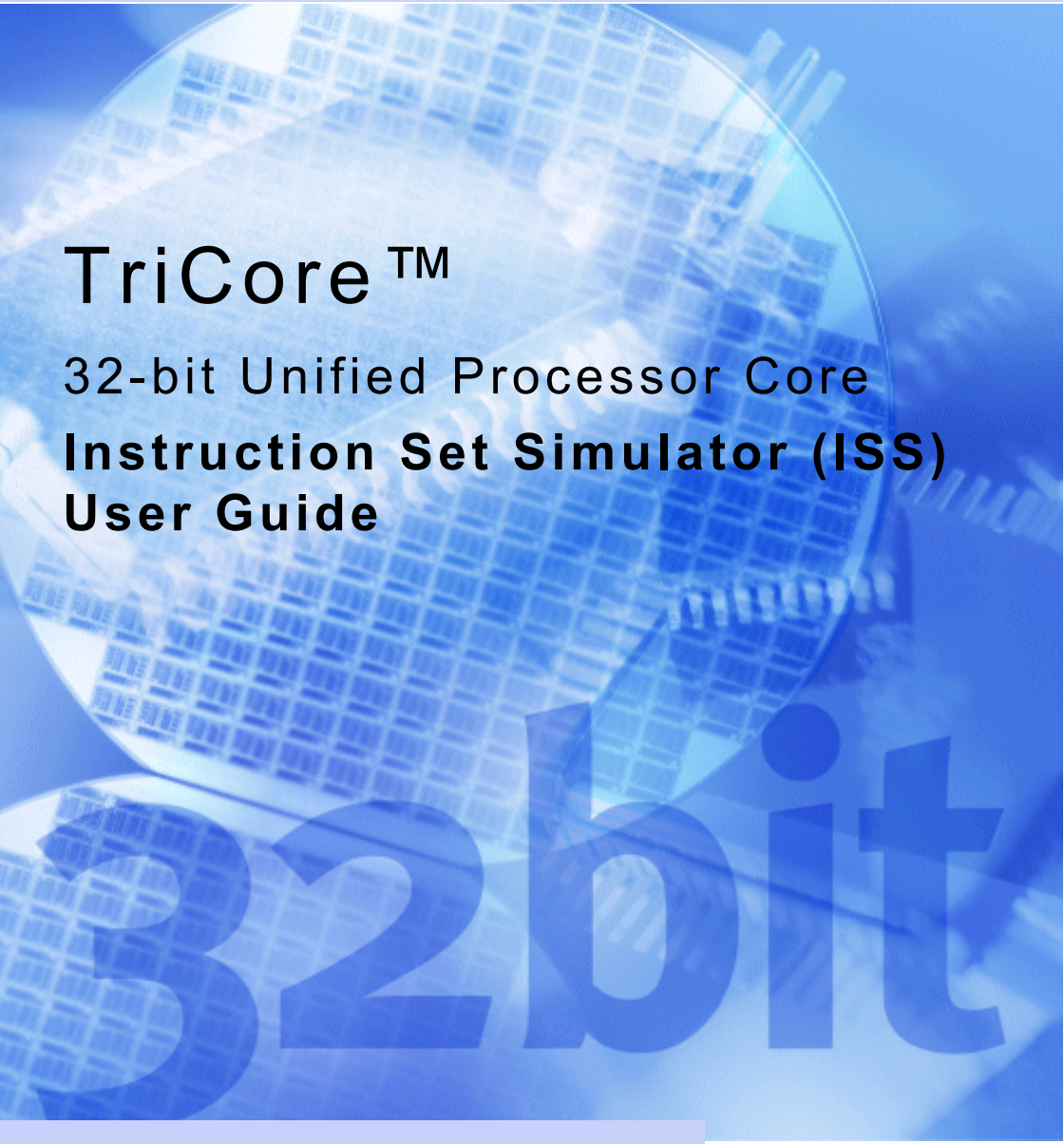
TriCore™

32-bit Unified Processor Core

Instruction Set Simulator (ISS) User Guide



Microcontrollers

Infineon technologies

Never stop thinking.

Edition 2005-01

Published by Infineon Technologies AG,
St.-Martin-Strasse 53,
D-81541 München, Germany

Attention please!

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Infineon Technologies is an approved CECC manufacturer.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office in Germany or our Infineon Technologies Representatives worldwide.

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

TriCore™

32-bit Unified Processor Core

Instruction Set Simulator (ISS) User Guide

Microcontrollers

Infineon technologies

Never stop thinking.

Previous Version

Version	Corresponding Tricore Models Release
V 1.0	1.4.4
V 1.1	1.4.5

We Listen to Your Comments

Is there any information in this document that you feel is wrong, unclear or missing? Your feedback will help us to continuously improve the quality of our documentation. Please send feedback (including a reference to this document) to:

ipdoc@infineon.com

1 Introduction

This document describes the TriCore™ Instruction Set Simulator (TSIM). TSIM is designed for analysis of the expected performance and code development of a user-defined implementation of the TriCore Core Instruction Set Architecture (ISA). TSIM has been integrated into the source-level debuggers included in this package. It can be installed on a Sun Microsystems SPARC workstation running Solaris or on a PC running Windows NT.

This document is written for application and system-level programmers and it assumes some familiarity with:

- C programming language or assembly language.
- Test program development.
- Microcontroller/microprocessor applications.
- Digital Signal Processing (DSP) architecture.
- Real-Time Operating System (RTOS).
- Analyzing chip/architecture performance using simulation.

This document is also available in electronic format (TSIM.pdf) on the CD-ROM supplied with TSIM. The TSIM.pdf can be displayed using the Adobe Acrobat Reader version 4.0.

1.1 CD-ROM Contents

- TriCore Instruction Set Simulator (TSIM)
- TriCore Instruction Set Simulator User's Guide (TSIM.pdf)
- Source-level debugger, including:
  - Compiler.
  - Assembler.
  - Linker.
  - Locator.
- Example files, including:
  - **PConfig** - external peripheral configuration file.
  - **MConfig** - memory configuration file.
  - **IConfig** - interrupt configuration file.
  - TriCore register address map.
  - DSP and RTOS programs, together with other program examples.

1.2 Additional Information

The following publications on the TriCore product line can be requested via your regional sales office, or visit the TriCore Internet pages **http://www.infineon.com/tricore**.

- TriCore Architecture Overview.
- TriCore Architecture Manual.
- Introducing TriCore (Brochure).
- TriCore Development Tools (Brochure).

2 Introducing TriCore™ Simulation

TriCore™ is the first single-core 32-bit microcontroller-DSP architecture optimized for real-time embedded systems. However with each successive semiconductor process technology, improvements in circuits, architecture, and logic design, simulators must be developed using increasingly complex modeling techniques. TSIM was developed to meet this challenge and provide the right tool for the job.

The TriCore architecture is well-supported by a robust set of hardware and software development tools such as TSIM, compiler-assembler tool chain, real-time operating systems, and third party source-level debuggers. Infineon worked closely with a number of third-party development tool partners to provide a comprehensive suite of development tools. These tools are bundled with TSIM, which has been integrated into our partners' source-level debugger.

This document focuses on using TSIM, the tool chain, and a source-level debugger to evaluate multiple, user-configurable implementations of the new TriCore family architecture.

2.1 Why Use Simulation?

It is common industry practice to evaluate the functionality and performance of new hardware and software by using development tools to study chip behaviour. Typically these tools are used for testing and debugging code and hardware at various stages in the development cycle.

There are some common problems that can best be solved with a simulator. There are also some jobs that may be impossible to do without a simulator. Simulators can best be used to study a problem in great detail by helping to determine what the code was doing at the point of failure. Single-stepping through the code essentially performs a dynamic and interactive code walk-through. If the problem can be reproduced in the simulator, it can be corrected.

A simulator can be used effectively in the early stages of software development, reducing the length of time spent later on system integration. Simulator access can also be provided to all members of a software team at relatively low cost compared to other instrumentation.

Simulation tackles the problems that involve collecting data about the run time behaviour of an embedded application. They are also beneficial for creating test conditions that can not easily be duplicated on real hardware. In the end, simulators help get a product to market faster and with fewer defects.

2.2 What Is TSIM?

TSIM provides a re-programmable simulation environment for configuring versions of a TriCore-based chip, including memory, interrupt mechanism, and peripheral address mapping. TSIM is primarily useful as a development tool in functional analysis, debugging, performance analysis, and trade-off analysis.

TSIM can be configured to evaluate different implementation approaches by changing memory parameters in the memory configuration file (**MConfig**) or to specify external interrupt events in the interrupt configuration file (**IConfig**). The peripheral configuration file (**PConfig**) is used to specify external peripherals that can interrupt the TriCore. The device configuration file **(DConfig)** is used to specify user defined device plugin modules. TSIM supports the Generic Debug Instrument (GDI) standard interface.

The following figure shows an overview of the simulation environment.

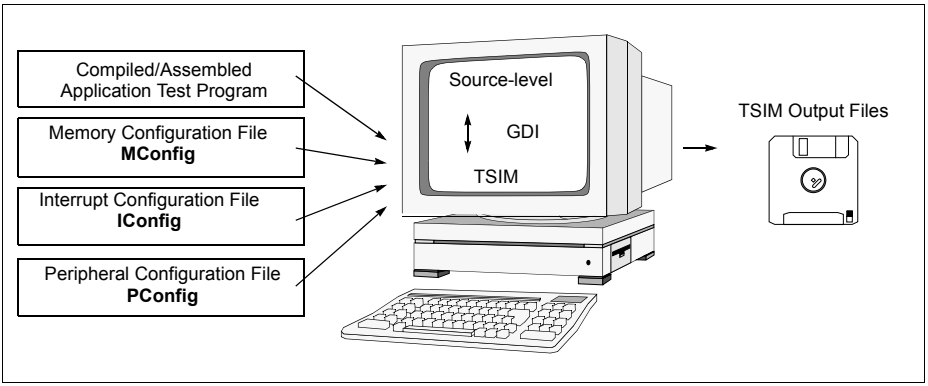


Figure 1 TSIM Simulation Environment

TSIM is an instruction-accurate model of a TriCore-based controller chip that is integrated into supported third party, source-level debuggers. Instruction-accurate means that TSIM sequentially executes program instructions without regard to pipeline behaviour, memory interface protocols, or bus protocols. One instruction is executed per “clock”. TSIM is designed as an algorithmic model of an example of the TriCore, and it also models the interrupt arbitration scheme.

TSIM is modeled at the register/instruction level. Therefore, the registers are modeled by the program variables, and instructions are modeled by program functions which operate on the register values.

2.3 TSIM Features

- Instruction-accurate simulator.
  - All TriCore instructions modeled in all instruction formats.
  - Traps and interrupts are modeled.
  - Load/stores are modeled.
  - Multiple issue and memory hierarchies are modeled.
- Configurable architecture.
  - Program boot startup address (MConfig).
  - Memory partitioning and latency parameters (MConfig).
  - External interrupt event parameters (IConfig).
  - Peripheral parameters (PConfig).
- Instruction/data memory.
  - Instruction memory is modeled as byte-addressable.
  - 4 GBytes of memory can be partitioned into different types with different access times.
  - Cache memory modeled only for timing information.
- Timing accuracy.
  - Number of cycles reported for user programs is an approximation.
  - Accuracy is dependent on application code type.
- High-speed simulations.
  - TSIM algorithm designed for high performance.
  - Each instruction decoded only once and then saved in buffer.
  - All ALU operations are modeled using function calls.
  - Table lookup for opcode specific information.
- TSIM Output File (**Sim.out**) displays:
  - The contents of input configuration files (MConfig, PConfig, IConfig, DConfig) used for this simulation.
  - Cache statistics (Hit/Miss ratios).
  - Number of instructions executed.
  - Number of seconds that the simulation ran.
  - Histogram of the instruction mix (total number of instructions and percent of total instructions).
  - Register file usage and display.

2.4 What Does TSIM Simulate?

TSIM simulates the first implementation of a TriCore-based chip, including program and data memories, the core, an interrupt controller, and several peripherals:

- General Purpose Timer (GPT) - used to time up to 8 programmed events.
- Watch Dog Timer (WDT) - used to recover from a hardware or software failure.
- System Timer (SYST) - provides global system time.

An example of a TriCore-based controller chip is shown below:

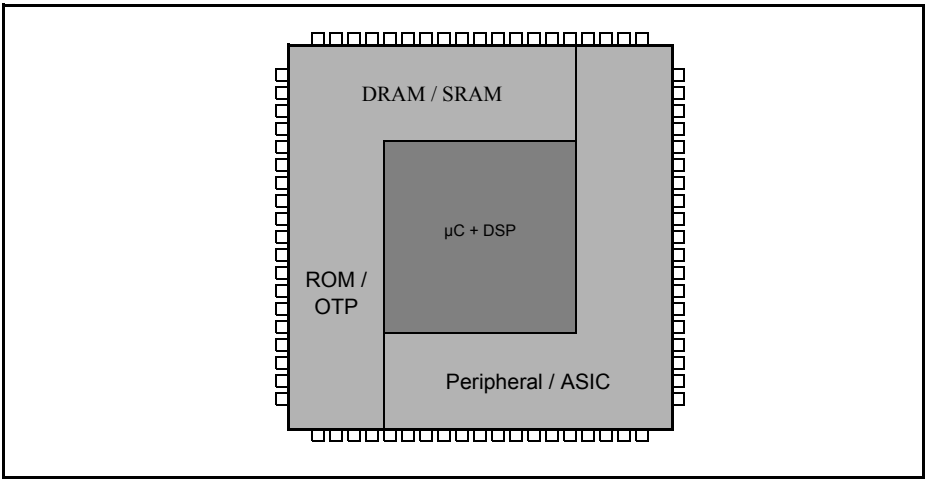


Figure 2 TriCore-Based Controller Chip Example

The following figure (**Figure 3**), shows an example of a TriCore based controller chip simulation model:

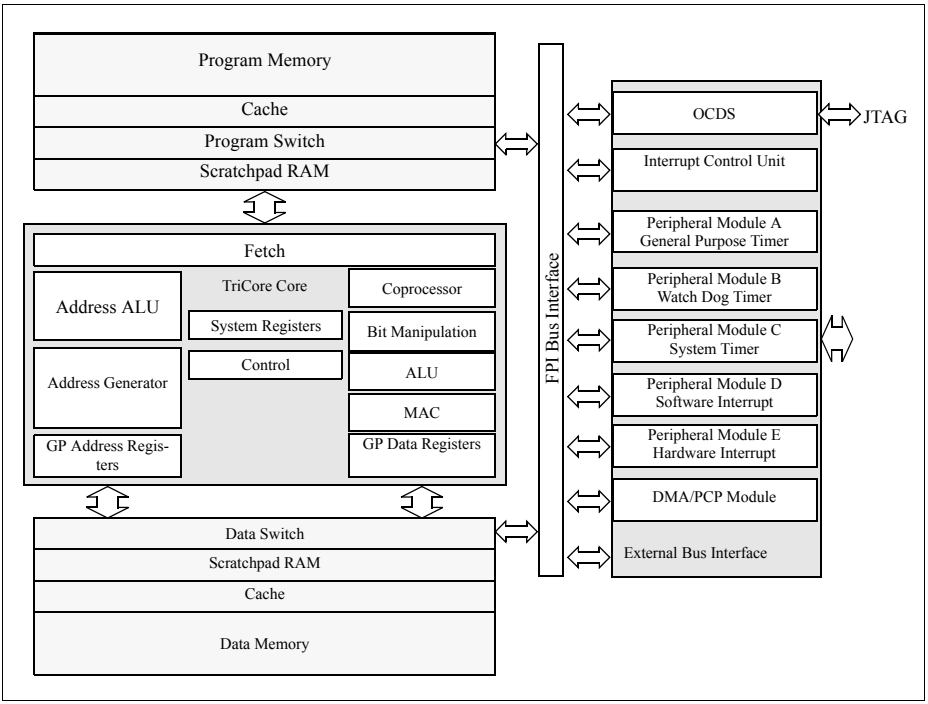


Figure 3 Example of a TriCore based Controller Chip Simulation Model

2.5 TriCore-Based Controller Chip Simulation Model

The TriCore architecture is a 32-bit load-store architecture. The definition of a word and the width of all registers is 32 bits. The instruction set supports operation on Booleans, bit strings, characters, signed fractions, addresses, signed and unsigned integers, and single-precision floating-point numbers. Most instructions operate on a specific data type while others are useful for manipulating several data types. Refer to the TriCore Architecture Manual for detailed information and the complete instruction set.

TSIM checks every memory access (read, write, or execute) for legality before performing the access. The checking is an in-range compare to a table entry, where the permission level and ranges are checked.

2.6 How Does TSIM Work?

TSIM is integrated into supported third-party, source-level debuggers, such as Altium's CrossView debugger and Greenhills Multi-debugger for example. The debuggers interact with the TSIM configurable input files and the test program to produce real-time data and simulation output files which contain simulation statistics. The debuggers are designed with an easy-to-use graphical user interface (GUI) that allows the user to halt a program's execution, examine the contents of core registers, addresses, and variables, and resume program execution. It runs on a SUN workstation under Solaris and on a PC running the Windows NT 4.0 operating system.

The command history and loadable symbols can be displayed, and command-line editing can be performed while running the debugger. Simulation can be interrupted by using the debugger to set breakpoints, display data, and then continue simulation. When the debugger is halted in this way, it is possible to:

- Examine the contents of the registers, caches, and memory and modify them if necessary.
- Review recent history of the program's flow of execution.
- View activity statistics for system-level functional blocks to measure and analyze overall performance.

Once a simulation terminates, TSIM creates an output file (Sim.out). These files contain the statistics of the last simulation and can be used to evaluate the performance and operation of multiple versions of a TriCore based chip (see **Chapter 4 Performance Analysis**).

2.6.1 Configurable Architecture (MConfig, IConfig, PConfig and DConfig)

TSIM allows configuration of different versions of the TriCore based chip through the use of the MConfig file to specify boot startup location and memory parameters. The IConfig file specifies the behaviour of external interrupt events, the PConfig file provides the information necessary to permit the user to simulate external peripherals or interrupt, while the device plugin mechanism(DConfig) allows the user to model his/her own devices/peripherials and for these to communicate via memory/interrupts with tsim itself. MConfig, IConfig, PConfig and DConfig are optional. If the MConfig file is missing, the default memory configuration is used.

See also **MConfig page 3-11**, **IConfig page 3-16**, **PConfig page 3-18**.

2.6.2 TSIM Accuracy

TSIM provides cycle counts which have been shown to be exact in most cases. The cycle counts are computed by examining the current instruction being executed and are based on latencies and instruction type. TSIM is supplied with information such as on-chip memory sizes and memory access times.

TSIM does accurately model the superscalar nature of two instructions (simultaneously executed) and counts the cycles correctly. TSIM does not model CPU stalls due to bus activity, numbered register file port limitations, or data contention.

3 Getting Started

This section describes how to get started using a source-level debugger with TSIM.

3.1 Test Configuration Guidelines

This section lists the general hardware and software guidelines needed to run the source-level debugger with TSIM. Please check your source-level debugger manual for specific configuration information.

3.1.1 Hardware and Software Platform

- Sun workstation running Solaris 2.7.

OR

- Windows NT 4.0 (or later) system with 24-32 MBytes RAM.

- Source-level debugger with TSIM installed.
- TSIM MConfig, IConfig, and PConfig input files (optional).
- Program source file.

3.1.2 Source-Level Debugger Features

Typical features of a supported source-level debugger (i.e. a debugger that includes TSIM) are shown in the following table (**Table 1**). The Windows NT version of the source-level debugger installs just like any other Windows NT application. For more specific installation instructions, please refer to the manual for your source-level debugger that was included in this package.

Table 1 Debugger Features

Feature	Capability
Multi-Window GUI	Keep track of information about registers, the stack and program variables.
Macros	Store and recall complex commands and expressions with a minimal number of keystrokes.
Data Breakpoints	Determine when memory addresses are accessed. With data breakpoints it is easy to track use and misuse of variables.
Code Breakpoints	Halt the program at critical junctions of program execution and observe values of important variables.

Table 1 Debugger Features (Continued)

Feature	Capability
Single Stepping	Run the program one line at a time to check dynamic values of variables and to investigate program flow. It is possible to single-step through high-level instruction and/or assembly code.
Assertions	Test for all sorts of error conditions throughout the length of the program.

3.1.3 Installing Input Configuration Files

To install the input configuration files, copy them from the CD-ROM to the directory specified in the environment variable, TSIM_CONFIG_DIR. The source-level debugger must also be run from this directory.

Note: If you do not want to use MConfig, IConfig or PConfig, move them out of the program directory or rename them.

3.2 Configuring TSIM Input Files

3.2.1 MConfig

MConfig contains the various memory parameters that TSIM will use for the current simulation.

The rules for the MConfig Format are:

- Each line should specify a unique memory part.
- Parameters must be listed in order.
- Each field must contain data.

Table 2 MConfig Keywords and Types of Parameters

Keyword	Param 1	Param 2	Param 3	Param 4	Comments
CACHE DATA	Data cache enable	size	#ways	latency	enable = 0 disable = 1
CACHE CODE	code cache enable	size	#ways	latency	enable = 0 disable = 1
SCRATCH DATA0	start addr	end addr	latency	n/a	end - start + 1 multiple of 8k
SCRATCH DATA1	start addr	end addr	latency	n/a	end - start + 1 multiple of 8k

Table 2 MConfig Keywords and Types of Parameters (Continued)

Keyword	Param 1	Param 2	Param 3	Param 4	Comments
SCRATCH CODE0	start addr	end addr	latency	n/a	end - start + 1 multiple of 8k
SCRATCH CODE1	start addr	end addr	latency	n/a	end - start + 1 multiple 8k
MEM1[1)]	start addr	end addr	latency	n/a	All other names
MEM2[2)]	start addr	end addr	latency	n/a	All other names are assumed to be general memory. Can use sequential numbers
__IOREAD	addr	n/a	n/a	n/a	Address reached before reading from STDIN
__IOWRIT E	addr	n/a	n/a	n/a	Address reached before writing to STDOUT
__CYCLE S	addr	n/a	n/a	n/a	Total number of cycles run up to specified address
BOOTADD RESS	addr	n/a	n/a	n/a	Specifies the boot address
EXITADDR ESS	addr	n/a	n/a	n/a	Specifies the exit address
SFR_BAS E_ADDRE SS	addr	n/a	n/a	n/a	Specifies the base address of the sfrs.
CPS_BAS E_ADDRE SS	addr	n/a	n/a	n/a	Specifies the base address of the cps registers
DMI_BASE _ADDRES S	addr	n/a	n/a	n/a	Specifies the base address of the dmi registers
PMI_BASE _ADDRES S	addr	n/a	n/a	n/a	Specifies the base address of the pmi registers
LFI_BASE _ADDRES S	addr	n/a	n/a	n/a	Specifies the base address of the lfi registers

Table 2 MConfig Keywords and Types of Parameters (Continued)

Keyword	Param 1	Param 2	Param 3	Param 4	Comments
WDT_BASE_ADDRESS	addr	n/a	n/a	n/a	Specifies the base address of the wdt registers
STM_BASE_ADDRESS	addr	n/a	n/a	n/a	Specifies the base address of the stm registers
GPTU0_BASE_ADDRESS	addr	n/a	n/a	n/a	Specifies the base address of the gptu0 registers

1) Not a keyword.

2) Not a keyword.

3.2.2 MConfig Example

The following sample MConfig has been included in this package. It is an ASCII text file and can be edited using any text editor. It is only an example to show syntax. If MConfig is used, no other memory is assumed other than what is in MConfig.

```
-----------------------------------------------------------------
BOOTADDRESS 0xD4000000
EXITADDRESS 0xD4000100
CACHE DATA 1 4 1
CACHE CODE 1 2 2
SCRATCH DATA0 0xD0000000 0xD0007fff 1
SCRATCH DATA1 0xD2000000 0xD2007fff 1
SCRATCH CODE0 0xD4000000 0xD4007fff 1
SCRATCH CODE1 0xD6000000 0xD6007fff 1
R  0xC0000000 0xC0001fff 3
R  0x80000000 0x8000ffff 3
R  0xA8000000 0xA000ffff 7
__IOREAD  0xa0000000
__IOWRITE 0xa0000004
__CYCLES  0xa0000008
SFR_BASE_ADDRESS 0xf7e10000
DMI_BASE_ADDRESS 0xf87ffc00
PMI_BASE_ADDRESS 0xf87ffd00
LFI_BASE_ADDRESS 0xf87fff00
WDT_BASE_ADDRESS 0xf0000000
STM_BASE_ADDRESS 0xf0000300
GPTU0_BASE_ADDRESS 0xf0000600
```

Explanation of MConfig example line-by-line:

BOOTADDRESS

- Specifies the address from which the core will start execution.

EXITADDRESS

- When the core reaches this address, execution will stop.

CACHE DATA 1 4 1

Specifies data cache enabled, 4 ways set associative, and 1-cycle latency.
If cache is specified, additional default values are listed in **Table 3**. Number of ways is set to 4 ways.STM_BASE_ADDRESS 0xf0000300

-

CACHE CODE 1 2 2

- Specifies code cache enabled, 2 ways set associative, and 2-cycle latency.
  If cache is specified, additional default values are listed in **Table 3**.

SCRATCH DATA0 0xD0000000 0xD0007fff 1

- Specifies data scratch pad 0 address range 0xD0000000 0xD0007fff with a 1-cycle latency.

SCRATCH DATA1 0xD2000000 0xD2007fff 1

- Specifies data scratch pad 1 address range 0xD2000000 0xD2007fff with a 1-cycle latency.

SCRATCH CODE0 0xD4000000 0xD4007fff 1

- Specifies code scratch pad 0 address range 0xD4000000 0xD4007fff with a 1-cycle latency.

SCRATCH CODE1 0xD6000000 0xD6007fff 1

- Specifies code scratch pad 1address range 0xD6000000 0xD6007fff with a 1-cycle latency.

R 0xC0000000 0xC0001fff 3

- Specifies memory address range 0xC0000000 0xC0001fff with a 3-cycle latency.

R 0x80000000 0x8000ffff 3

- Specifies memory address range 0x80000000 0x8000ffff with a 3-cycle latency.

R 0xA8000000 0xA000ffff 7

- Specifies memory address range 0xA8000000 0xA000ffff with a 7-cycle latency.

__IOREAD 0xa0000000

- Specifies the address that when reached by the program counter, the system reads from standard in (STDIN) to data register 5 in the TriCore register file.

__IOWRITE 0xa0000004

- Specifies the address that when reached by the program counter, the system writes to standard out (STDOUT) from data register 5 in the TriCore register file.

__CYCLES 0xa0000008

– Specifies the address that when reached by the program counter, the system stores the total number of cycles run so far into register 5 in the TriCore register file.

```
*_BASE_ADDRESS 0xf0000300
```

– specifes the base address for a particular set of registers.

3.2.3 Default MConfig Parameters

If an MConfig file is not used or an error is detected in the given MConfig file, then tsim will use a default memory configuration. The exact details of this differ dending on whether one is using tsim or tsim2. These are detailed in the Sim.out file.

3.2.4 IConfig

TSIM contains a programmable interrupt generator that allows programming the occurrence of any of the following:

- A single interrupt.
- A repeating interrupt.
- A repeating interrupt jittered by a random number of cycles.
- A random interrupt.

Up to 256 interrupts can be configured in the TriCore architecture. The programmable interrupt generator uses the parameters specified in IConfig to configure external interrupts at specific cycles during simulation (**Table 3**). This feature can also be used by those creating software applications to program interrupts for software test purposes.

Note: Before using interrupts and peripherals for simulation, they must first be initialized in your program. Refer to the TriCore Architecture Manual for more information.

The rules for IConfig file format are as follows:

- Each line should specify a unique interrupt event.
- More than one interrupt format can be used to specify the interrupt events.
- Parameters must be listed in order.
- Each field must contain data.
- Keywords are shown in the following table (**Table 3**).
- There are no default parameters for IConfig.
- If IConfig is used, a corresponding PConfig must also be used.

Table 3 IConfig Keywords and Types of Parameters

Keyword	Param1	Param2	Param3	Param4	Param5	List
ONESHOT	cycle num	number of interrupts	n/a	n/a	n/a	SRNs[1)]
Comment: Single interrupt.						
FIX_INTERVAL	start cycle	end cycle	freq	number of interrupts	n/a	SRNs
Comment: Repeating interrupt.						
JITTERS	start cycle	end cycle	freq	jitter amt	number of interrupts	SRNs
Comment: Repeated interrupt fluctuating by random number of cycles in jitter ranges.						
RANDOM	start cycle	end cycle	number of interrupts	n/a	n/a	SRNs
Comment: Single random interrupt.						

1) Service Request Node (SRN) specifies priority of the interrupt.

3.2.5 IConfig Example

The following sample IConfig file is included in the package. It is an ASCII text file that can be edited with any text editor:

```
-----------------------------------------------------------------
ONESHOT 100 1 6
FIX_INTERVAL 100 200 10 2 6 7
JITTER 0 200 10 5 3 6 7 8
RANDOM 300 350 4 6 7 8 9
-----------------------------------------------------------------
```

Explanation of IConfig example line-by-line:

`ONESHOT 100 1 6`

- Fire a ONESHOT interrupt, starting at 100 cycles for 1 peripheral whose SRN number is 6.

`FIX_INTERVAL 100 200 10 2 6 7`

- Fire a FIX_INTERVAL interrupt starting at 100 cycles, ending at 200 cycles; fire that interrupt every 10 cycles between that cycle range for 2 peripherals whose SRN numbers are 6 and 7 respectively.

`JITTER 0 200 10 5 3 6 7 8`

- Fire a JITTER interrupt starting at 0 cycles and ending at 200 cycles; fire that interrupt 10 cycles between that cycle range with a jitter range of 5(+/-5) for 3 peripherals whose SRN numbers are 6, 7, and 8 respectively.

`RANDOM 300 350 4 6 7 8 9`

- Fire a RANDOM interrupt starting at 300 cycles and ending at 350 cycles for 4 peripherals whose SRN numbers are 4, 6, 7, and 9 respectively.

3.2.6 PConfig

PConfig is used to specify the address of SRNs for external devices that can interrupt the TriCore processor. This mechanism uses SRN numbers to identify each peripheral.

IConfig and PConfig are interrelated (**Figure 4**). PConfig specifies the SRN number and address for each peripheral, and IConfig specifies an external interrupt behaviour for a particular peripheral by using its SRN number.

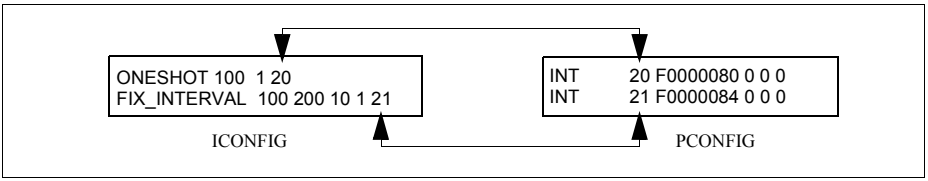


Figure 4 IConfig & PConfig Relationship

Note: In order to use interrupts and peripherals for simulation, they must first be initialized and enabled in the program using the Service Request Control (SRC) register. Refer to the TriCore Architecture Manual for more information.

The rules for the PConfig file format are as follows:

- Each line should specify a unique SRN number and peripheral.
- Up to 256 peripherals can be specified.
- SRNs listed in **Table 4** are reserved for internal peripherals.
- No two peripherals can have the same SRN address.
- The peripheral address must be word-aligned.

Table 4 Internal Peripherals

Name	SRN	Reserved	Comment
WDTCON	418	0 0 1	WDTCON (F0000380 is used) at address F000001C
WDTTIM	419	0 0 2	WDTTIM (F0000384 is used) at address F0000018
GTSRC0	8	0 0 0	
GTSRC1	7	0 0 0	
GTSRC2	6	0 0 0	
GTSRC3	5	0 0 0	
GTSRC4	4	0 0 0	
GTSRC5	3	0 0 0	
GTSRC6	2	0 0 0	

Table 4 Internal Peripherals (Continued)

Name	SRN	Reserved	Comment
GTSRC7	1	0 0 0	
CPUSRC0	17	0 0 0	
CPUSRC1	16	0 0 0	
CPUSRC2	15	0 0 0	
CPUSRC3	14	0 0 0	
PCPSRC0	12	0 0 0	
PCPSRC1	11	0 0 0	
PCPSRC2	10	0 0 0	
PCPSRC3	9	0 0 0	
SBSRC0	13	0 0 0	
SSC0ESRC	382	0 0 0	
SSC0RSRC	381	0 0 0	
SSC0TSRC	380	0 0 0	
S1TBSRC	372	0 0 0	
S1ESRC	371	0 0 0	
S1RSRC	370	0 0 0	
S1TSRC	369	0 0 0	
S0TBSRC	359	0 0 0	
S0ESRC	358	0 0 0	
S0RSRC	357	0 0 0	
S0TSRC	356	0 0 0	

3.2.7 PConfig Example

A sample PConfig file has been included in this package. It is an ASCII text file and can be edited with any text editor.

```
------------------------------------------------------------------
INT  20 F0000088 0 0 0
INT  21 F0000084 0 0 0
INT  22 F0000088 0 0 0
INT  23 F000008C 0 0 0
INT  45 F0000090 0 0 0
INT  57 F0000094 0 0 0
INT  61 F0000098 0 0 0
------------------------------------------------------------------
```

Explanation of PConfig example, column by column:

`INT`

– Interrupt. Software can change the priority of the interrupt, enable an interrupt, or write to the service control register in order to modify the interrupt's behaviour (see the **IConfig Example** for an alternative method).

`20`

– SRN number. This must be unique. Valid range from 19 to 255.

`F0000088`

– Peripheral address. External peripherals must reside in segment "E" of memory.

`0`

– Reserved.

`0`

– Reserved.

`0`

– Reserved.

3.2.8 DConfig

The DConfig (device configuration) file is a way of allowing tsim to function in conjuction with user defined peripherals/devices. Tsim includes a device plugin mechanism. In essence, this means that a number of hooks are provided to enable a user to create a dynamically linked library to model a peripheral of his/her chosing, and as long as it respects the provided interface, it can intercept bus traffic, read/write external memory, interrupt the core, etc. Incuded in the release package is an example device plugin. For more details on the nature of both the device plugin interface and the callback interface into the core, see the example device source code provided with this release. This will also give details on the building of such libraries, and the mecnahism used to tell tsim where to find plugin modules (i.e. the DConfig file).

3.2.9 GPT External Stimulus

TSIM provides a second way to program interrupts using the GPT. This method mimics use of the 8 external GPT pins on the chip.

The GPT has 8 service request nodes which correspond to each of 8 external inputs. The timers within the GPT may be triggered from the internal clock or from one of the 8 external pins. If the user wishes to cause a timer to count external transitions, they must provide a separate data file named stimulus_0. This file contains one data line per simulated clock cycle. The data line consists of 1's and 0's, each representing one of 8 external input pins.

Since this data file could be quite long in the case of an extensive test (such as running an operating system with applications for example), it is recommended that it be program generated. The user should note that it is inadvisable to trigger interrupts of any kind before the interrupt vector pointer (BIV), is programmed or the interrupt handlers are loaded into memory.

If the stimulus_0 file is not presented to TSIM, the following error message will be printed on the test.out file:

```
Warning(gpt): Can't open stimulus file "stimulus_0" for read.
```

This is a non-fatal message informing the user that no stimulus_0 file was supplied.

3.2.10 GPT External Stimulus Example

```
;in0   in1   in2   in3   in4   in5   in6   in7   in8
0      0     0     0     0     0     0     0     0
0      1     0     1     0     0     1     0     0
0      0     0     0     0     0     0     0     0
1      1     1     1     1     1     1     1     1
0      0     0     0     0     0     0     0     0
0      1     0     0     0     0     0     0     0
```

The example illustrates that multiple SRNs can be triggered simultaneously. It is important to note that each line of the stimulus_0 file represents a set of external inputs to be applied during a single clock. The interrupts will be triggered in the order specified by the Service Request Priority Number (SRPN) the user has programmed into the GPT's SRNs.

Depending upon how the user has programmed the GPT control registers, a particular timer may be programmed to count rising edges or rising and falling edges (quadrature mode) on one of the in0 -> in7 external pins. An interrupt may be generated when a timer rolls over or reaches zero (count-down mode) depending on control register state written into the control register by the user. The timers within GPT may also be programmed to run from the internal clock within TSIM. Refer to the Application Note, "Programming the GPT".

3.3 Test Program Overview

This section provides a brief overview of a test program. Refer to the source-level debugger manual for more specific instructions and guidelines on developing test programs. The following diagram illustrates the code sections found in a typical test program.

System startup code should initialize:

- Data memory.
- Core registers.
- Interrupt/trap handler code.
- Enable/disable interrupts.
- Context save areas.
- Protection memory areas.
- Peripheral service control registers.

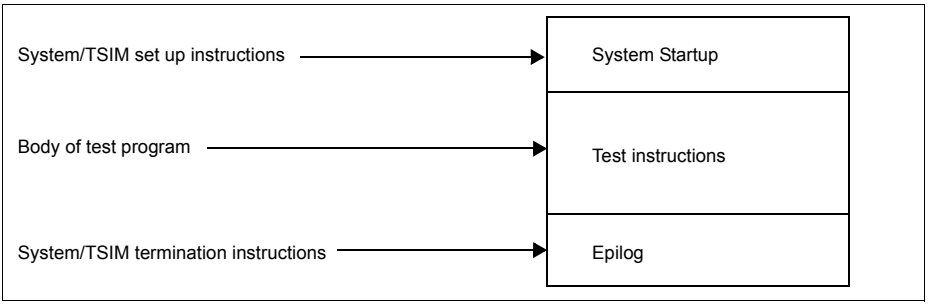


Figure 5 Test Program Code Sections

3.4 Test Program Tool Chain

Once a test program has been written, it must be compiled or assembled, linked and then loaded into the source-level debugger. The Compiler, Assembler, Linker and Locator programs are part of the source-level debugger's tool chain. The following figure (**Figure 6**) shows a typical example of a tool chain.

Instructions for running tool chain programs will vary depending on the source-level debugger. Refer to the manual included with the source-level debugger.

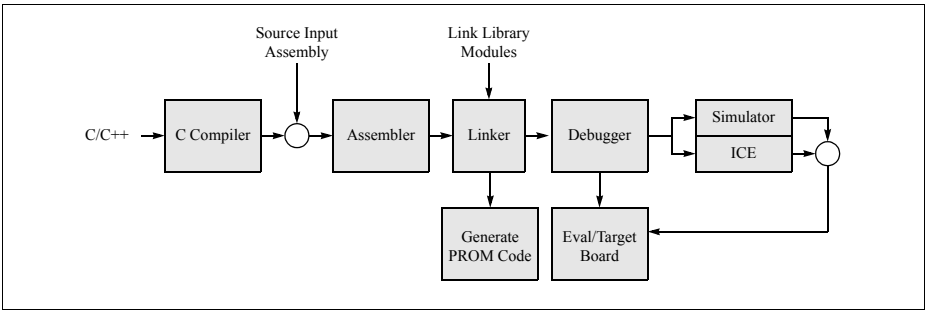


Figure 6 Overview of a Typical Tool Chain

3.5 Starting The Simulation

Simulation is started by invoking the source-level debugger. Refer to the debugger documentation to find out the specific instructions needed to load the compiled test program and to start simulation.

3.5.1 During Simulation

Use these basic simulation features to help develop and debug the program:

- Set breakpoints at critical sections in the test program. When a breakpoint is encountered check registers, variables, and addresses.
- Single step through the program to check registers, variables and addresses after each instruction has executed.
- When an interrupt event occurs check registers, variables and addresses.

3.6 Stopping Simulation

Like starting simulation, stopping simulation is specific to the source-level debugger. Refer to the debugger documentation for steps needed to terminate simulation and to exit the debugger.

3.6.1 After Simulation

When simulation is terminated, TSIM creates an output file (Sim.out) that can be used for analyzing system performance.

4 Performance Analysis

4.1 Overview

By tracking the performance of the application code, measurements can be performed on sections of code at any stage of the project. This can determine whether a software implementation of specific functionality has enough performance to meet the application requirements.

Evaluating and analyzing the performance of a TriCore-based chip is usually an iterative process (See **Figure 7**). At each step simulation information is gathered and interpreted, and the application code is refined accordingly. This process can be repeated as often as necessary to arrive at an acceptable approach, or to compare the merits of several different configurations of a TriCore-based chip.

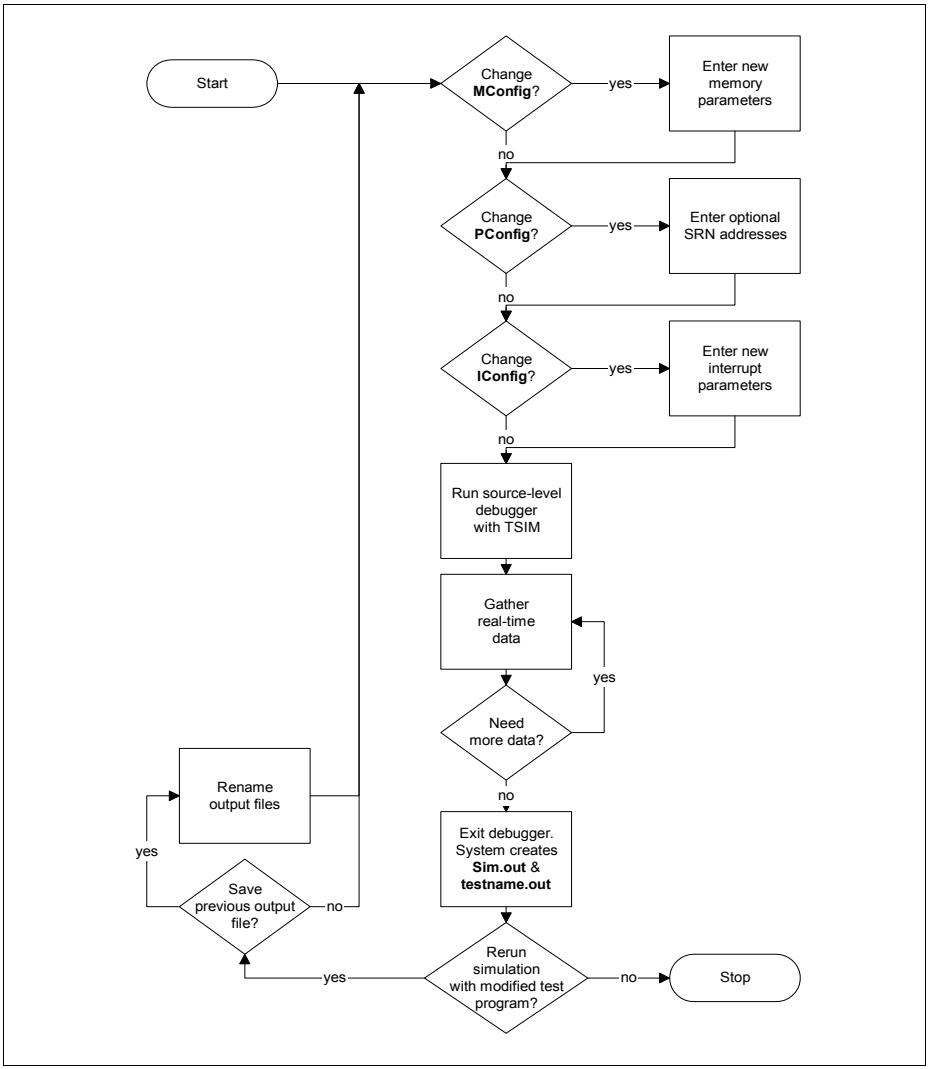


Figure 7 Overview of Performance Analysis Process

4.2 Real-time Data

During simulation, more detailed analysis may be carried out on a particular section of the program by using breakpoints. By setting breakpoints, simulation can be interrupted at the first breakpoint; variables, registers, and addresses can be examined and simulation can then be resumed until the second breakpoint is reached. During simulation, statistical information can be printed to display program behaviour between breakpoints (i.e. the section under study).

4.2.1 Simulation Suggestions

System Performance

At some point in the development cycle, the system will need to be tuned to achieve the required performance levels. TSIM collects the data needed to analyze TriCore performance. Examples of data that can be collected from simulation include which functions in the code get the most CPU cycles, so optimization time can be invested where it will do the most good, or where a redesign can improve throughput. For TriCore-based chips with on-chip cache, realigning code and data can dramatically affect performance. It is also possible to measure the cache hit ratio and then rearrange memory contents to minimize cache misses, and to get the best cache performance possible.

TSIM models certain features of TriCore which are used to determine the system performance accurately:

- Each instruction has a fixed latency.
- Parallel execution of instructions.
- Cache data/code with LRU replacement.
- Branch prediction.
- Variable memory configuration and latency.
- Loop instruction modeling.

Trace

An instruction trace can be enabled at the start of the code sequence of interest. Information about the sequence of instructions executed can be obtained from the trace.

Profile

Profiling code is useful to get information on how much time is being spent in a certain memory code range.

Interrupts/Traps

It is important to be able to specify an interrupt and observe what happens when the interrupt occurs. For example, a simulation can be run until the beginning of the Interrupt Service Routine (ISR), followed by a step to the point where the second interrupt is desired, and then the simulator can be told to assert an interrupt immediately. By continuing with single-stepping through the code, it is possible to observe what happens to the processor, and to ensure that control gets properly transferred to the ISR.

Debugging ISRs is one of the most powerful features of TSIM. TSIM has the ability to generate an interrupt on demand by using either application code or IConfig. For example, an interrupt can be made to occur every 100,000 cycles, and then use the TSIM output files to verify that the interrupt occurred when specified.

Aside from interrupts, there are other types of traps that can occur at run time in embedded systems. TSIM allows the user to ensure that these conditions are handled properly. For example, it is important to know if code tries to execute an invalid opcode, or if an arithmetic overflow occurs in an integer calculation. A handler for these exceptions may be desirable.

4.3 TSIM Output Files

When the debugger is terminated, TSIM automatically generates an output file (Sim.out) that contain statistics from the current simulation. They are text files that can be edited and imported into any database program for historical analysis. These statistics are cleared automatically when the debugger is restarted. Therefore, in order to save the previous output files, the debugger must be terminated and the files renamed before restarting the debugger. The output files can also be printed to review statistics collected during the current and previous simulations. This approach can be useful in gaining insight into a program's overall behaviour.

Below is an example of Sim.out. This reports the memory used, interrupt specified, and ISC register information. This simulation included a test program, an IConfig interrupt input file, and an MConfig memory input file. The simulation statistics then follow.

```
----------------------------------------------------------------
Sim.out
----------------------------------------------------------------
INFINEON TECHNOLOGIES TRICORE ISS

ATTENTION: TSIM is an ARCHITECTURAL MODEL
cycle counts are approximate.

No memory (MConfig) file specified/Error opening: ./MConfig

Default Cache Configuration:
```

```
TSIM CODE CACHE enabled: 0, size: 8192, line size: 32, ways: 2,
access time: 0
TSIM DATA CACHE enabled: 0, size: 2048, line size: 16, ways: 2,
access time: 0

Default Memory Configuration:
CODE SCRATCH: DEFAULT: 0xD4000000 - 0xD4007FFF, access_time: 1
DATA SCRATCH: DEFAULT: 0xD0000000 - 0xD0007FFF, access_time: 1
EXTERNAL MEM: DEFAULT: 0x80000000 - 0xCFFFFFFF, access_time: 7
EXTERNAL MEM: DEFAULT: 0xDF000000 - 0xDFFFFFFF, access_time: 7
EXTERNAL MEM: DEFAULT: 0xE2000000 - 0xE2000FFF, access_time: 7
EXTERNAL MEM: DEFAULT: 0xDE000000 - 0xDEFFFFFF, access_time: 7
EXTERNAL MEM: DEFAULT: 0xF0000000 - 0xF0003FFF, access_time: 7

No interrupt (IConfig) file specified/Error opening: ./IConfig
No peripheral (PConfig) file specified/Error opening: ./PConfig
No  device  plugin  (DConfig)  file  specified/Error  opening:
./DConfig

RESET PC = a0000000
All Files setup for simulation run
.....................................................
Downloading program to Simulator
Reset and initialize Simulator


RESET PC = a0000000
Start Simulation

In ExecSingleStep
Simulation stopped due to PWRDN
In ExecStop
In GdiClose
sim done closing up ....
Simulation complete

*************** Simulation statistics ***************

Total number of instructions executed = 6969
Total number of cycles run            = 67648
Total number of seconds for execution = 0      seconds

total number of interrupts fired=0
```

```
No Dcache accesses

No Ccache accesses
Instruction histogram:                %      (Num of instructions)
          MAC:              0%        (4)
          JMPS:            18%        (1304)
          LOADS:           19%        (1348)
          STORES:          15%        (1068)
          ARITHMETIC:      46%        (3210)
          SYS:          0%        (35)
          MISC:         0%        (0)

Data Register usage for 32 bit instructions:

DR[00] = 1029     DR[01] = 2201   DR[02] = 148   DR[03] = 113
DR[04] = 109     DR[05] = 33   DR[06] = 2   DR[07] = 0
DR[08] = 56     DR[09] = 22   DR[10] = 10   DR[11] = 7
DR[12] = 3     DR[13] = 0   DR[14] = 470   DR[15] = 1056

Address Register usage for 32 bit instructions:

AR[00] = 3     AR[01] = 3   AR[02] = 433   AR[03] = 2127
AR[04] = 680     AR[05] = 611   AR[06] = 89   AR[07] = 17
AR[08] = 3     AR[09] = 3   AR[10] = 488   AR[11] = 69
AR[12] = 78     AR[13] = 119   AR[14] = 480   AR[15] = 712

Data Register usage for 16 bit instructions:

DR[00] = 717     DR[01] = 2081   DR[02] = 114   DR[03] = 53
DR[04] = 43     DR[05] = 42   DR[06] = 2   DR[07] = 0
DR[08] = 110     DR[09] = 85   DR[10] = 79   DR[11] = 80
DR[12] = 74     DR[13] = 68   DR[14] = 71   DR[15] = 943

Address Register usage for 16 bit instructions:

AR[00] = 0     AR[01] = 0   AR[02] = 588   AR[03] = 1081
AR[04] = 1128     AR[05] = 1075   AR[06] = 97   AR[07] = 3
AR[08] = 0     AR[09] = 0   AR[10] = 422   AR[11] = 68
AR[12] = 121     AR[13] = 164   AR[14] = 88   AR[15] = 386
```

4.3.1 Total Number of Instructions Executed

This data can be used to make initial rough performance comparisons between different implementation approaches (i.e. different versions of the TriCore core). For example, two versions of MConfig are configured for a particular test program, each using a different optimization approach. By running each of them through TSIM, the total instruction counts provide a quantitative comparison of the approaches.

Note: For these types of comparisons, a program's code size may bear little or no relation to instruction count. Even a small program may have a very large instruction count if its execution involves lots of looping. Conversely, execution need not pass through every possible branch every time a program is run, so that the instruction count could appear smaller than the code size.

4.3.2 Total Number of Cycles Run

Because of the superscalar nature of the TriCore architecture, it is possible that the number of instructions executed are higher than the number of cycles run. However, memory latencies can influence this behaviour, depending on the parameters used in MConfig.

4.3.3 Total Number of Seconds for Execution

This number is approximately the time it took (in seconds) to execute the current simulation on the host machine.

4.3.4 Total Number of Interrupts Fired

This number is the total number of interrupt events that occurred in the current simulation. It is possible to compare this number with the number of interrupt events specified in IConfig. If the number of interrupts fired is 0 and you configured them in IConfig, then you may not have enabled or initialized interrupts in your test program.

4.3.5 Data Cache Hit Rate

Programs vary widely in the extent and efficiency of their cache use, so the relationship between performance and cache size must be characterized carefully. Caches can account for a significant proportion of die area and therefore of final product cost. Striking the appropriate balance between the application's performance requirements and cost constraints is one of the most important trade-offs in a design.

If a cache shows only a small number of references relative to the overall instruction count, or has a very high hit rate (95% or better), try reducing the cache's size in MConfig. If overall performance is still adequate at the lower cache size, the die size can be reduced or the free area reassigned to some more useful function.

Conversely, if either cache shows a significant number of references and a hit rate below 90 to 95%, it is worth considering some combination of options (such as an increase in cache size for example) to reduce the associated performance penalty.

4.3.6 Instruction Histogram

The instruction mix data provides a first impression of a program's behaviour, which is useful in determining those aspects of the system than are constraining the program's overall performance. For example, a section of code that performs very few multiply/divide operations would not gain much benefit from the inclusion of a faster multiply/divide unit.

4.3.7 Data Register Usage for 32-bit Instructions

The output file lists the data register usage for 32-bit instructions register-by-register. For example:

```
Data Register usage for 32 bit instructions:

DR[00] = 5436
```

This means that data register 00 was accessed 5436 times for 32-bit instructions.

4.3.8 Address Register Usage for 32-bit Instructions

The output file lists the address register usage for 32-bit instructions register-by-register.

4.3.9 Data Register Usage for 16-bit Instructions

The output file lists the data register usage for the 16-bit instructions register-by-register.

4.3.10 Address Register Usage for 16-bit Instructions

The output file lists the address register usage for 16-bit instructions register-by-register

M

O

P

R

S

T

W

http://www.infineon.com

Published by Infineon Technologies AG