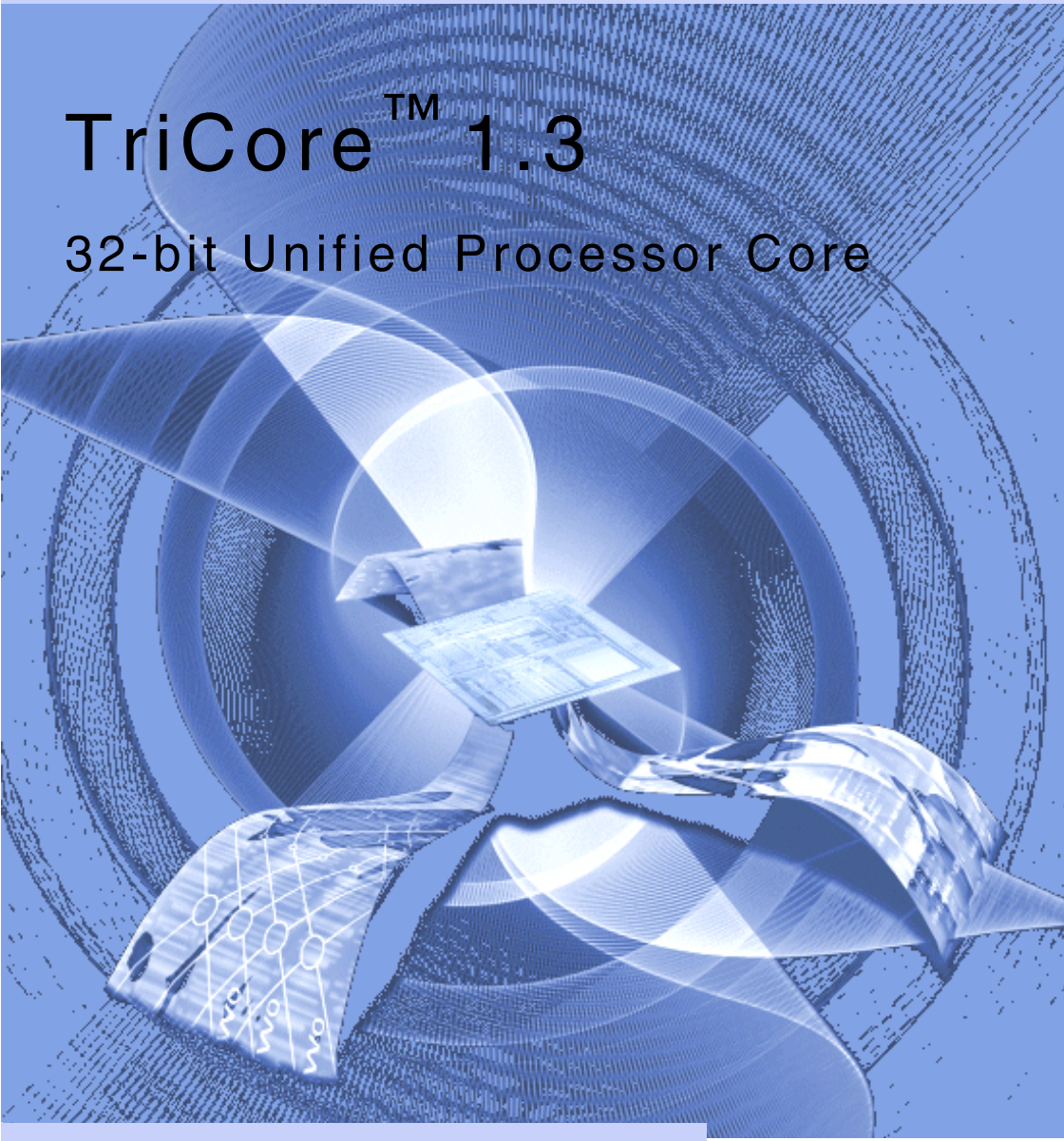
TriCore™ 1.3

32-bit Unified Processor Core



IP Cores

Infineon technologies

Never stop thinking.

Edition May 2002

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

© Infineon Technologies AG 2002.
All Rights Reserved.

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 (see address list).

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™ 1.3

32-bit Unified Processor Core

IP Cores

Infineon technologies

Never stop thinking.

TriCore™ 1.3 Architecture Overview Handbook

Revision History:	**May 2002**	V1.3.3
Previous Version:	-	
Page	Subjects (major changes since last revision)	
All	Document rewrite for latest issue of TriCore core	

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

ipdoc@infineon.com

Table of Contents Page

Table of Contents **Page**

1 Preface

This document has been written for Engineering Managers and hardware/software Engineers, to provide an overview of the TriCore Instruction Set Architecture (ISA).

TriCore is Infineon's architecture for a unified MCU/DSP processor core. This architecture is available today in two implementations:

- TriCore version 1.3 (v1.3 or TC1.3)
- TriCore version 2.0 (v2.0 or TC2)

Each version of the architecture has a complete set of documentation of its own: Product Brief, Architecture Manual, Data Sheet and Integration Manual.

More information about the TriCore product line can be found in the following documents:

- TriCore Architecture Manual
- TriCore Instruction Set Simulator User's Guide
- TriCore Development Tools (brochure)

These documents are available from your regional sales office, or visit the TriCore internet Home Page to download PDF versions.

The TriCore home page is:

- **http://www.infineon.com/tricore**

Details of your nearest Infineon sales office can be found at:

- **http://www.infineon.com/business/offices/index1.htm**

2 TriCore Family Architecture

2.1 Overview

Present and future trends for embedded systems include a convergence of microcontroller and DSP architectures, as well as super-integration of memory and logic. Embedded applications are evolving towards a single System-On-a-Chip (SOC) comprising of a unified microcontroller-DSP core (32 bits), data and program memory (RAM, ROM, OTP, etc.), and custom application-specific logic (ASIC):

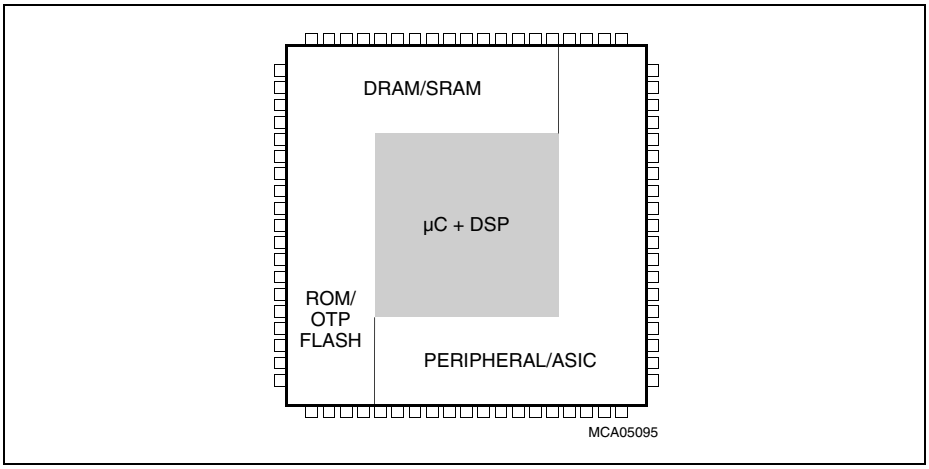


Figure 1 System-on-a-Chip for Embedded Applications

The single core provides virtual multiprocessing, eliminating the need for multiple controllers and DSPs. On-chip memories enhance performance and reduce system power dissipation, while the integration of system peripherals and customer-specific logic increases the overall system performance at a reduced cost. This single core scenario is imperative for embedded systems, as more and more applications demand higher system performance at a reasonable price. With cost-effective processor performance, more work can be off-loaded from hardware to software tasks running on these powerful, multi-tasking CPUs.

The resident (‘off-the-shelf’) Real-Time Operating System (RTOS) has a compact kernel with appropriate plug-ins for debug and communications for example. The application layer on top of the RTOS is automatically generated with the help of App-builder programs that draw on rich library routines such as DSP, floating-point and peripheral management.

The elements for tomorrow's embedded systems exist today. The TriCore Instruction Set Architecture (ISA) from Infineon is the first single-core 32-bit microcontroller-DSP architecture optimized for real-time embedded systems.

TriCore unifies the best of 3 worlds: the real-time capabilities of microcontrollers, the computational power of DSPs and the highest performance/price features of a RISC load/store architecture onto a compact, re-programmable core.

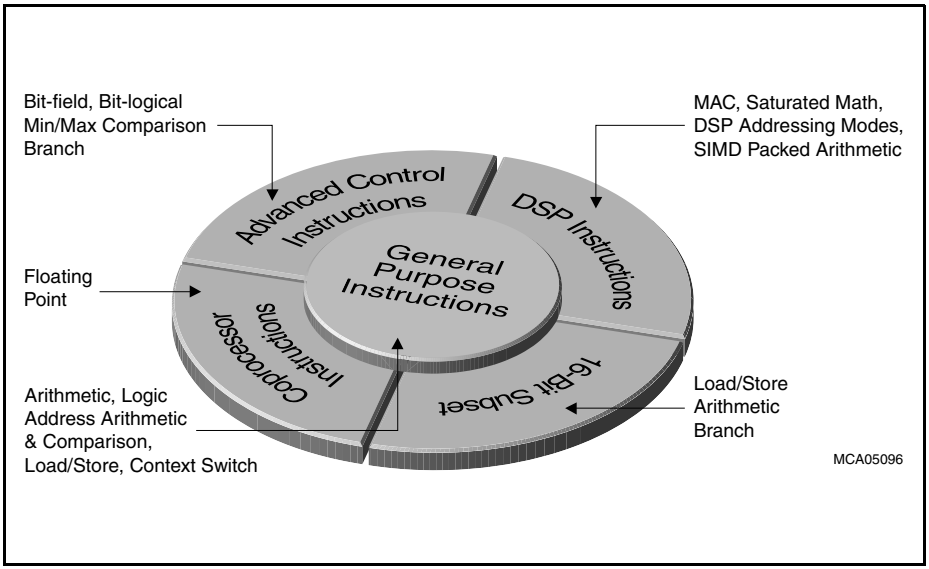


Figure 2 TriCore Architecture

TriCore is supported by Infineon's comprehensive library of peripheral module options (such as DMA and Debug), while both the type (SRAM, DRAM, ROM, FLASH or OTP) and size of on-chip memory are configurable. The core and peripherals are easily connected to yield a high-performance, cost-effective System-On-a-Chip (SOC), tailored to individual applications.

The key benefits to using the TriCore for a real-time embedded system are:

- The single architecture merges both DSP and microcontroller features without sacrificing the performance of either.
- Fast task switching (via an internal wide bus to on-chip memory) allows TriCore to be used effectively as a virtual multiprocessor. It can switch for example, from a DSP to a microcontroller task in two cycles.
- Large on-chip memory blocks (RAM, ROM, DRAM, OTP, FLASH) result in higher performance, more reliable operation, and reduced system power consumption.
- The architecture allows direct control of on-chip peripherals without additional glue logic. TriCore supports a lean but powerful memory protection and on-chip debug support scheme.
- A freely intermixed 16-bit and 32-bit instruction format reduces code size for an application by approximately 30 to 40%.
- Interrupts are processed as injected calls and are handled by the same mechanism.

TriCore uses a RISC-like register model and load/store architecture to support HLL (High-Level Language) compilers and their optimization strategies. Fast context switching and low interrupt latencies enable a flexible distribution of processor performance among concurrent tasks and effective control of peripheral events, while integrated debug hardware eases the software development cycle.

The TriCore architecture can automatically save or store half the register context upon an interrupt within two cycles. The architecture therefore provides fast interrupt response without the need for major housekeeping before entering the real interrupt service routine. The ISA is also capable of interacting with different system architectures, including multiprocessing. This flexibility at the implementation and system levels allows different trade-off's between performance and cost at any point in time.

The native microcontroller-DSP capabilities of the architecture allow the microcontroller and DSP performance of each TriCore core to be tuned by software. For example, the performance of a 300-MHz TriCore-1.3 core with a sustained 450MIPS rating is 280 microcontroller MIPS + 170 DSP MIPS, or 200 microcontroller MIPS + 250 DSP MIPS, depending on how the system designer implements load-sharing in software.

2.1.1 Feature Summary

The key features of the TriCore Instruction Set Architecture (ISA) are:

- 32-bit architecture
- 4-Gbyte virtual or physical data, program and input/output (I/0) address space
- 16-bit and 32-bit instructions for reduced code size
- Low interrupt latency with fast context switch using wide pathway to on-chip memory
- Dedicated interface to application-specific co-processors to allow the addition of customised instructions
- Zero overhead loop capabilities
- Dual single-clock-cycle 16 x 16 multiply-accumulate unit (with optional saturation)
- Optional Floating-Point Unit (FPU) and Memory Management Unit (MMU)
- Extensive bit handling capabilities
- Single Instruction Multiple Data (SIMD) packed data operations
- Flexible interrupt prioritization scheme
- Byte and bit addressing
- Little-endian byte ordering for data memory and CPU registers
- Memory protection
- Debug support

2.2 TriCore Instruction Categories

TriCore architecture offers a flexible set of instruction formats to optimize code space. Although the architecture is 32 bits, there are 16-bit instruction formats available to code the most frequently required instructions in a reduced amount of memory space. This reduces the instruction code space by an average of one third or more, compared to conventional RISC architectures.

TriCore instructions are subdivided into the following categories.

- Branch
- Load/Store
- System
- 16-Bit Subset
- Arithmetic (Integer, DSP and SIMD Packed Arithmetic)
- Comparison
- Bit Manipulation
- Address Arithmetic and Address Comparison

Note: See ***"Instruction Set Highlights" on Page 22*** *for more details.*

2.3 Target Applications

TriCore has been optimized to meet the requirements of embedded applications such as computer peripherals, automotive power-train controllers, vehicle dynamics systems, cellular communications and networking equipment.

An increasing number of embedded designs employ both a microcontroller or microprocessor and a DSP or hard-wired ASIC, but a single TriCore device can replace both of these components because of its inherent microcontroller-DSP capabilities and its ability to switch between those tasks at breakneck speed.

3 TriCore Programming Model

This section discusses those aspects of the TriCore architecture that are visible to software: the supported data types and formats, the various addressing modes that the architecture provides and the memory model.

3.1 Architectural Registers

The TriCore architectural registers (**Figure 3**) consist of 32 General-Purpose Registers (GPRs), two 32-bit registers with program status information (PCXI and PSW), and a Program Counter (PC).

Four GPRs have special functions:

- D15 is used as an implicit data register
- A10 is the stack pointer (SP)
- A11 is the return address register
- A15 is the implicit base address register

PCXI, PSW, and PC are Core Special Function Registers (CSFRs). The PCXI and PSW registers contain status flags, previous execution and protection information.

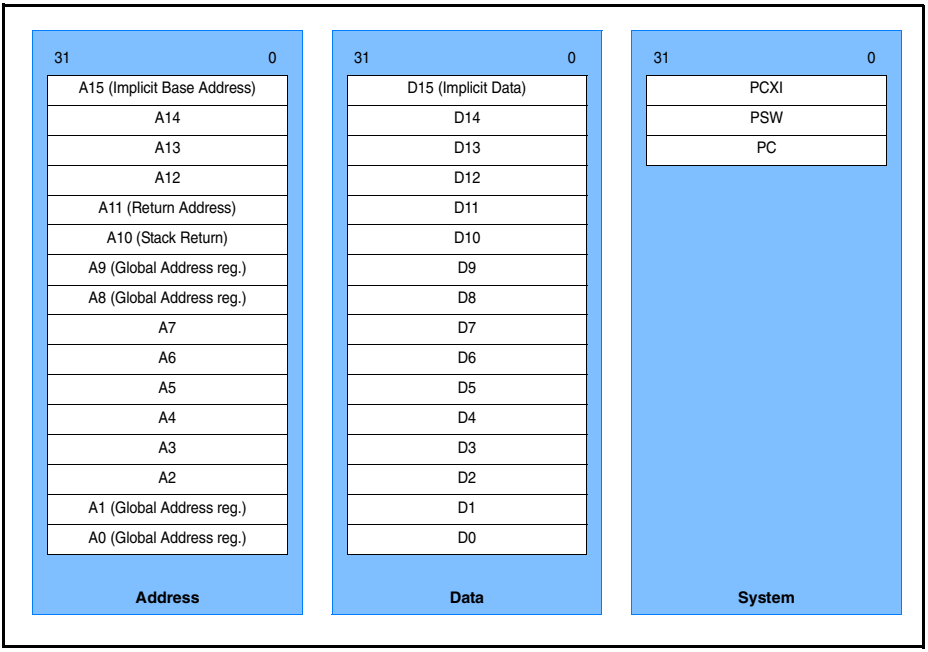


Figure 3 Architectural Registers (GPRs)

3.2 Data Types and Formats

The TriCore instruction set supports operations on booleans, bit strings, characters, signed fractions, addresses, signed and unsigned integers, and single-precision floating-point numbers. Most instructions work on a specific data type, while others are useful for manipulating multiple data types.

- Boolean
- Address
- Bit String
- Signed/Unsigned Integer
- Character
- Signed Fraction
- IEEE-754 single-precision floating-point

The General-Purpose Registers (GPRs) are all 32-bits wide, and most instructions operate on word (32-bit) values. This means that when data with fewer bits than a word is loaded from memory, it must be sign or zero-extended before operations can be applied to the full word. The sign or zero extension is carried out concurrently, as part of the load operation.

The data memory and CPU registers store data in little-endian byte order (the least-significant bytes are at lower addresses). Little-endian memory referencing is used consistently for data and instructions. When the TriCore system is connected to an external big-endian device, translation between big- and little-endian format is performed by the bus interface.

Alignment requirements are different for addresses and data. Addresses (32-bits) must be aligned on a word boundary to permit transfers between address registers and memory. For transfers between data registers and memory, data may be aligned on any halfword boundary, regardless of size. Bytes may be accessed in any valid byte address with no alignment restrictions.

3.3 Memory Model

The TriCore architecture can access up to 4 Gbytes of unified program and I/O (Input/ Output) memory. The address width is 32-bits.

The address space is divided into 16 regions or segments (0 through 15). Each segment is 256-Mbytes.

The upper four bits of an address select the specific segment. The first 16-Kbytes of each segment can be accessed using either absolute addressing or absolute bit addressing with the bit set and bit clear instructions.

Figure 4 shows the TriCore architecture's address space mapping.

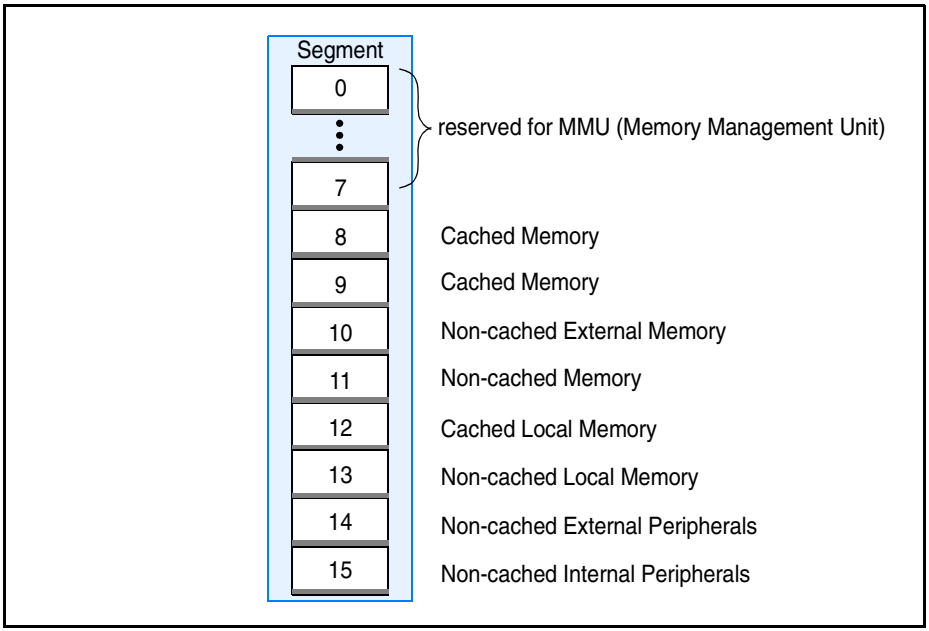


Figure 4 Address Map & Memory Model

3.4 Addressing Modes

Addressing modes allow load and store instructions to efficiently access simple data elements within data structures, such as records, randomly and sequentially accessed arrays, stacks and circular buffers. Simple data elements are 8, 16, 32 or 64 bits wide.

The TriCore architecture supports seven addressing modes (**Table 1**). These addressing modes offer efficient compilation of C/C++ programs, easy access to peripheral registers and the efficient implementation of typical DSP data structures (circular buffers for filters and bit-reversed indexing for FFTs).

Table 1 TriCore Architecture Addressing Modes

Addressing Mode	Address Register Use	Offset Size (bits)
Absolute	None	18
Base + Short Offset	Address Register	10
Base + Long Offset	Address Register	16
Pre-increment	Address Register	10
Post-increment	Address Register	10
Circular	Address Register Pair	10
Bit-reverse	Address Register Pair	—

Addressing modes which are not directly supported in the hardware can be synthesized through short instruction sequences using indexed addressing, PC-relative addressing, or extended absolute addressing.

4 Tasks and Contexts

A **Task** is an independent thread of control. There are two types of task: **Software-Managed Tasks** (SMTs) and **Interrupt Service Routines** (ISRs).

Software-managed tasks are created through the services of a real-time kernel or OS, and are dispatched under the control of scheduling software. SMTs are sometimes referred to as ‘user’ tasks, assuming that they will execute in user mode. Interrupt Service Routines are dispatched by hardware in response to an interrupt. In this architecture ISR only refers to the code that is invoked directly by the hardware.

Each task is allocated its own permission level, depending on the task’s function. Individual permissions are enabled/disabled primarily through the IO mode bits in the Processor Status Word (PSW).

Associated with any task are a set of state elements known collectively as the task’s Context. The Context is everything the processor needs in order to define the state of the associated task and enable its continued execution. It includes the CPU general registers that the task uses, the task’s program counter (PC), and its Program Status Information (PCXI and PSW). The TriCore architecture efficiently manages and maintains the tasks’ contexts through hardware.

The Context is subdivided into the **Upper Context** and the **Lower Context**.

The **Upper Context** consists of the upper address registers A10 - A15, and the upper data registers D8 - D15. These registers are designated as non-volatile, for purposes of function calling. The upper context also includes PCXI and PSW.

The **Lower Context** consists of the lower address registers A2 through A7, and the lower data registers D0 through D7, together with the PC.

Registers A0 and A1 in the lower address registers and A8 and A9 in the upper address registers are defined as **System Global Registers**. These registers are not included in either Context partition, and are not saved or restored across calls or interrupts. They are normally used by the operating system normally to reduce system overhead.

The TriCore architecture uses linked lists of fixed-size **Context Save Areas** (CSAs). A CSA is 16 words of on-chip memory storage, aligned on a 16-word boundary. Each CSA can hold exactly one upper or one lower context. CSAs are linked together through a Link Word.

The TriCore architecture saves and restores context much more quickly than conventional microprocessors and microcontrollers. Its unique memory subsystem design with a wide data path, allows the TriCore architecture to perform rapid data transfers between processor registers and on-chip memory.

Context switching occurs when an event or instruction causes a break in program execution, which then results in the CPU needing to resolve this event before continuing with the program.

The events and instructions which will cause a break in program execution are:

1. Interrupt or service requests
2. Traps
3. Function calls

5 Interrupt System

5.1 Overview

A key feature of the TriCore architecture is its powerful and flexible interrupt system. The interrupt system is built around programmable **Service Request Nodes** (SRNs).

A **service request** is defined as an interrupt request or a DMA (Direct Memory Access) request. A service request may come from an on-chip peripheral, external hardware or software.

Conventional architectures handle service requests by loading a new Program Status (PS) from a vector table in data memory. With the TriCore architecture however, service requests jump to vectors in code memory. This procedure reduces response time for service requests. The first instructions of the Interrupt Service Routine (ISR) execute at least three cycles earlier than they would otherwise.

5.2 Interrupt Priority

Service Requests are prioritized to enable nested interrupts. The rules for prioritization are:

- A Service Request can interrupt the servicing of a lower priority interrupt.
- Interrupt sources with the same priority cannot interrupt each other.
- The Interrupt Control Unit (ICU) determines which source will win arbitration based on the priority number.

All Service Requests are assigned Priority Numbers (SRPNs). Even the CPU has its own priority number. Different service requests must be assigned different priority numbers.

The maximum number of interrupt sources is 255. Programmable options range from one priority level with 255 sources, up to 255 priority levels with one source each.

Interrupt numbers are assumed to be assigned in linear order of interrupt priority. This is feasible because interrupt numbers are not hardwired to individual sources, but are assigned by software executed during the power-on boot sequence.

The interrupt examples on the following page provide illustrations of three Task 1 interruptions.

5.3 Interrupt Examples

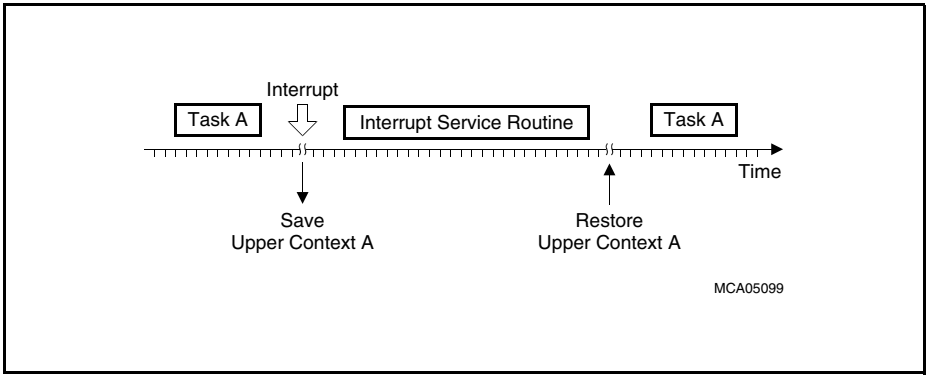


Figure 5 Simple Interrupt

For a simple interrupt, TriCore automatically saves the Upper Context on entering the Interrupt Service Routine (ISR). The Upper Context Registers can then be used within the ISR. When the Return from Execution instruction is issued, the Upper Context from the time of the interrupt is automatically restored.

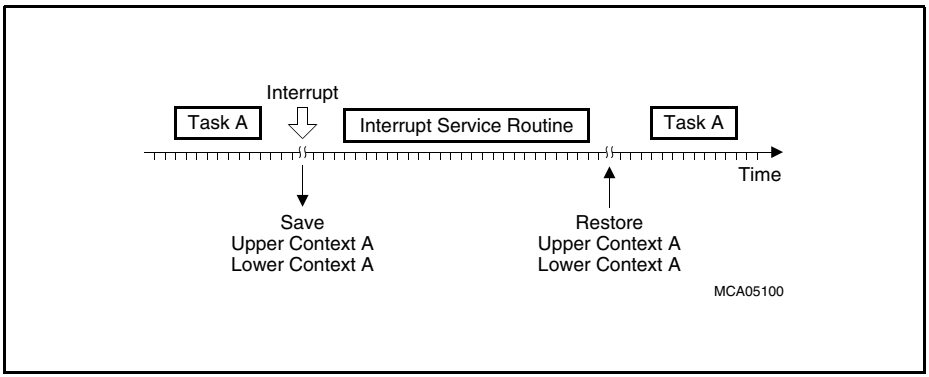


Figure 6 General Interrupt

In the general interrupt the Upper Context is automatically stored. The ISR explicitly saves the Lower Context using the SVLCX instruction. Both Upper and Lower Context Registers can be used within the rest of the ISR. Before returning to Task 1, the Restore Lower Context instruction (RSLCX) is issued, followed by a return from exception (RFE). This automatically restores the Upper Context.

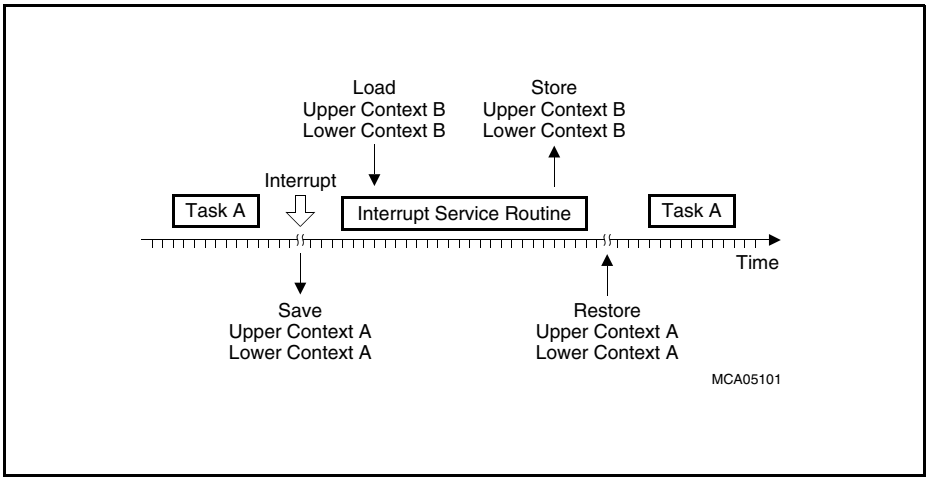


Figure 7 Simple Interrupt with Context Switch

In the ISR, explicit Upper and Lower Context values are loaded from memory using the LDUCX and LDLCX instructions. These values were saved from a previous call or interrupt for explicit use in the ISR. At the end of the ISR, new values to be used in a subsequent ISR call are stored explicitly using the STUCX and STLCX instructions.

6 Trap System

A trap occurs as a result of an event such as a Non-Maskable Interrupt (NMI), an instruction exception, or illegal access for example.

The TriCore architecture contains eight trap classes. These traps are further classified as synchronous or asynchronous, hardware or software. Each trap is assigned a Trap Identification Number (TIN) that identifies the cause of the trap within its class.

The eight trap classes are:

- MMU (Memory Management Unit)
- Internal Protection
- Instruction Errors
- Context Management
- Assertion
- System Bus & Peripheral Errors
- System Call
- NMI (Non-Maskable Interrupt)

7 Protection System

The Protection System is used to assign access permissions to memory regions for data and code. These Protection capabilities are used to protect core system functionality from bugs that may have slipped through testing.

TriCore's Protection System provides the essential features to isolate errors, and protects critical system functions against both software and transient hardware errors.

TriCore's embedded architecture allows each task to be allocated the specific permission level it needs to perform its function. The three permission levels are:

- **User-0 Mode**
  - for tasks that do not access peripheral devices.
- **User-1 Mode**
  - for tasks that access common, unprotected peripherals. Interrupts can be disabled for a short period at this level.
- **Supervisor Mode**
  - permits read/write access to system registers and protected peripheral devices.

The memory protection model for the TriCore architecture is based on address ranges, where each address range has an associated permission setting.

Address ranges and their associated permissions are specified in two to four identical sets of tables residing in Core SFR (CSFR) space. Each set is referred to as a **Protection Register Set** (PRS).

When the Protection System is enabled, TriCore checks every load/store or instruction fetch address for legality before performing the access. To be legal, the address must fall within one of the ranges specified in the currently selected PRS, and permission for that type of access (read, write, execute) must be present in the matching range.

7.1 Memory Management Unit (MMU)

TriCore can also make use of an optional Memory Management Unit (MMU).

From MMUs standpoint, TriCore's memory space has the following characteristics:

- Two addressing modes: physical or virtual (where physical attributes override virtual attributes).
- 4-GByte physical address space.
- 4-GByte virtual address space.

When the MMU is present, the protection will depend on whether TriCore is in virtual or physical mode:

- In physical mode TriCore native protection system is used.
- In virtual mode (when the lower 7 segments are used) the MMU is active.

8 Instruction Set Highlights

This section provides high-level details on the TriCore Instruction Set. Complete information on all instructions can be found in the TriCore Architecture Manual.

8.1 Instruction Set Summary

Note: Shaded entries indicate 16-bit instructions.

Table 2 TriCore Instruction Set Summary

Mnemonic	Definition
ABS	Absolute value
ABSDIF	Absolute value of difference
ABSDIFS	Absolute value of difference with saturation
ABSS	Absolute value with saturation
ADD	Add
ADDC	Add carry
ADDI	Add immediate
ADDIH	Add immediate high word
ADDS	Add with saturation
ADDSC	Add scaled address
ADDX	Add and generate carry
AND	Logical AND
AND.comp	Compare, AND and accumulate
AND.logic	Bit and logical accumulate
ANDN	Logical AND Not
BISR	Begin ISR
BMERGE	Merges even/odd
BSPLIT	Split in even/odd
CACHEA.I	Cache Address Invalidate
CACHEA.W	Cache Address Writeback
CACHEA.WI	Cache Address Writeback and Invalidate
CADD	Conditional ADD
CADDN	Conditional ADD Not
CALL	Call
CALLA	Call absolute
CALLI	Call indirect

Table 2 TriCore Instruction Set Summary (continued)

Mnemonic	Definition
CLO	Count leading ones
CLS	Count leading signs
CLZ	Count leading zeros
CMOV	Conditional move
CMOVN	Conditional move Not
CSUB	Conditional subtract
CSUBN	Conditional subtract Not
DEBUG	Debug
DEXTR	Double extract
DISABLE	Disable interrupt
DSYNC	Synchronize data
DVADJ	Divide adjust
DVINIT	Divide initialization word
DVSTEP	Divide step
ENABLE	Enable interrupt
EQ	Equal
EQANY	Multiple compare
EQZ	Equal zero address
EXTR	Extract bit field
GE	Greater than or equal
IMASK	Insert mask
INS	Insert bit
INSERT	Insert
INSN	Insert bit Not
ISYNC	Synchronize instructions
IXMAX	Finds maximum value in signed array
IXMAX.U	Finds maximum value in unsigned array
IXMIN	Finds minimum value in signed array
IXMIN.U	Finds minimum value in unsigned array
J	Jump unconditional
JA	Jump unconditional absolute
JEQ	Jump if equal
JGE	Jump if greater than or equal

Table 2 TriCore Instruction Set Summary (continued)

Mnemonic	Definition
JGEZ	Jump if greater than or equal to zero
JGTZ	Jump if greater than zero
JI	Jump indirect
JL	Jump and link
JLA	Jump and link absolute
JLEZ	Jump if less than or equal to zero
JLI	Jump and link immediate
JLT	Jump if less than
JLTZ	Jump if less than zero
JNE	Jump if not equal
JNED	Jump if not equal and decrement
JNEI	Jump if not equal and increment
JNZ	Jump if not equal to zero
JZ	Jump if zero
LD	Load
LDLCX	Load lower context
LDMDST	Load modify store
LDUCX	Load upper context
LEA	Load Effective address
LOOP	Loop
LOOPU	Loop unconditional
LT	Less than
MADD(S)	Multiply-Add (S = with Saturation)
MADDM(S).H	Packed Multiply-Add Q Format - Multiprecision
MADDR(S).H	Packed Multiply-Add Q Format w/ Rounding
MADDR(S).Q	Multiply-Add Q Format with Rounding
MADDSU(S).H	Packed Multiply-Add/Sub Q Format
MADDSUM(S).H	Packed Multiply-Add/Sub Q Format - Multiprecision
MADDSUR(S).H	Packed Multiply-Add/Sub Q Format w/ Rounding
MAX	Maximum value
MFCR	Move from Core Register
MIN	Minimum value
MOV	Move

Table 2 TriCore Instruction Set Summary (continued)

Mnemonic	Definition
MOVH(.A)	Move halfword to address
MSUB(S)	Multiply-Subtract (S = with Saturation)
MSUBAD(S).H	Packed Multiply-Sub/Add Q Format
MSUBADM(S).H	Packed Multiply-Sub/Add Q Format - Multiprecision
MSUBADR(S).H	Packed Multiply-Sub/Add Q Format w/ Rounding
MSUBM(S).H	Packed Multiply-Subtract Q Format - Multiprecision
MSUBR(S).H	Packed Multiply-Subtract Q Format w/ Rounding
MSUBR(S).Q	Multiply-Subtract Q Format w/ Rounding
MTCR	Move to Core Register
MUL(S)	Multiply (S = with Saturation)
MUL(S).U	Multiply Unsigned (S = with Saturation)
MUL.H	Packed Multiply Q Format
MUL.Q	Multiply Q Format
MULM.H	Packed Multiply Q Format - Multiprecision
MULR.H	Packed Multiply Q Format with Rounding
MULR.Q	Multiply Q Format with Rounding
NAND	Logical NAND
NE	Not equal
NEZ.A	Not equal zero address
NOP	No operation
NOR	Logical NOR
NOT	Bitwise complement
OR	Logical OR
OR.comp	Compare, OR and accumulate
OR.logic	Bit OR logical accumulate
ORN	Logical OR Not
PACK	Translates in floating-point format
PARITY	Computes parity
RET	Return from call
RFE	Return from Exception
RSLCX	Restore lower context
RSTV	Reset overflow flags
RSUB	Reverse subtract

Table 2 TriCore Instruction Set Summary (continued)

Mnemonic	Definition
RSUBS	Reverse subtract with saturation
SAT	Saturate result
SEL	Select
SELN	Select Not
SH	Shift
SH.comp	Compare accumulate and shift
SH.logic	Bit shift logical accumulate
SHA	Arithmetic shift
SHAS	Arithmetic shift with saturation
ST	Store
STLCX	Store lower context
STUCX	Store upper context
SUB	Subtract
SUBC	Subtract with carry
SUBS	Subtract signed with saturation
SUBX	Subtract extended
SVLCX	Save lower context
SWAP	Swap
SYSCALL	System call
TLBDEMAP	Uninstall a mapping in the MMU
TLBFLUSH	Flush mappings from MMU
TLBMAP	Install a mapping in the MMU
TLBPROBE.A	Probe the MMU for a virtual address
TRAPSV	Trap on sticky overflow
TRAPV	Trap on overflow
UNPACK	Translates from floating-point format
XNOR	Logical exclusive NOR
XOR	Logical exclusive OR
XOR.comp	Compare, XOR and accumulate

8.2 16-bit and 32-bit Instructions

The TriCore architecture supports both 16- and 32-bit instruction formats.

All instructions have a 32-bit format, but the 16-bit instructions are a subset of the 32-bit instructions, chosen because of their frequency of use and included to reduce code space.

The 16-bit instructions employ one or more of the following methods to allow encoding in 16 bits:

- 2-operand alternative to 3-operand ALU instructions (destination = second source operand)
- Implicit source, destination, or base address operand
- Small constants
- Short branch displacements
- Short load/store offsets

Note: The width of the address/data is implicit in the opcode.

8.3 Load and Store Instructions

The load and store instructions move data (8,16, 32 or 64 128 bits) between registers and memory, using the seven addressing modes shown in **Table 1**.

The addressing mode determines the effective byte address for the load or store instruction and any update of the base pointer Address register.

8.4 Arithmetic Instructions

Arithmetic instructions operate on data and addresses in registers. Status information about the result of the arithmetic operations is recorded in five status flags. These instructions are further categorized into integer arithmetic, DSP arithmetic and packed arithmetic instructions.

8.4.1 Integer Arithmetic

Move

- MOV sign-extends the value to 32 or 64 bits
- MOV.U zero-extends to 32 bits
- MOVH loads a 16-bit constant into the most-significant 16 bits of the register and zero fills the least-significant 16 bits

Addition and Subtraction

- ADD no saturation
- ADDS signed saturation

– ADDS.U unsigned saturation
– ADDX extended precision addition
– ADDC Add with Carry
– ADDI Add Immediate
– ADDIH Add Immediate High Word

Because the large immediate of ADDI is sign-extended, it may be used for both addition and subtraction.

The RSUB (Reverse Subtract) instruction subtracts a register from a constant. Using zero as the constant yields negation as a special case.

Multiply and Multiply-Add

Multiplication of two 32-bit integers that produce a 32-bit result can be handled using:

– MUL (Multiply Signed)
– MULS (Multiply Signed with Saturation)
– MULS.U (Multiply Unsigned with Saturation).

The following instructions produce the full 64-bit result, which is stored to a register pair:

– MULM (Multiply with Multiword Result) used for signed integers
– MULM.U (Multiply with Multiword Result Unsigned) used for unsigned integers.

Note: Special multiply instructions are used for DSP operations.

– MADD (Multiply-Add instruction)

The multiply-add instruction (MADD) multiplies two signed operands, adds the result to a third operand, and stores the result in a destination. Because the third operand and the destination do not use the same registers, the intermediate sums of a multi-term, multiply-add instruction can be saved without requiring any additional register moves.

– MADDS (Multiply-Add with Saturation)
– MADDS.U (Multiply-Add with Saturation Unsigned)

The MADD, MADDS (Multiply-Add with Saturation), and MADDS.U (Multiply-Add with Saturation Unsigned) instructions operate on and produce 32-bit integers; MADDS and MADDS.U will saturate on signed and unsigned overflow, respectively.

– MADDM (Multiply-Add with Multiword Result)
– MADDM.U (Multiply-Add with Multiword Result Unsigned)
– MADDMS (Multiply-Add Multiword with Saturation)
– MADDMS.U (Multiply-Add Multiword with Saturation Unsigned)

MADDM, MADDM.U, MADDMS and MADDMS.U can be used to add the 64-bit product to a 64-bit source and produce a 64-bit result.

– MSUB (Multiply-Subtract instruction)

The set of Multiply-Subtract (MSUB) instructions, which supports the accumulation of products using subtraction instead of addition, provides the same set of variations as the MADD instructions.

Division

– DVADJ (Divide Adjust)
– DVINIT (Divide Initialization)

The core ISA for TriCore has no direct support for division. However, all implementations to date have included ISA extensions in the form of instructions for a tightly-coupled co-processor to support divide operations.

Division is a multi-cycle operation - to reduce interrupt lockout time, division is performed in steps. One divide step operation develops eight bits of quotient in four cycles, for current implementations. The number of steps can be varied, depending on the width of the quotient expected.

Setup for a sequence of divide steps is accomplished with a DVINIT instruction, while a DVADJ (divide adjust) performs a final adjustment for negative quotients.

Absolute Value, Absolute Difference

– ABS (Absolute Value)
– ABSDIF (Absolute Difference)

The ABS and ABSDIF instructions compute the absolute value of a signed number or absolute value of the difference between two signed numbers, respectively. Each instruction has a version that saturates when the result is too large to be represented as a signed number.

Min, Max, Saturate

– MIN (Minimum)
– MAX (Maximum)

The MIN and MAX instructions calculate the minimum or maximum value between two operands, respectively.

– SAT (Saturate)

The SAT instructions saturate the result of a 32-bit calculation before storing it in a byte or halfword in memory or a register.

Conditional Instructions

- CADD (Conditional Add)
- CADDN (Conditional Add Not)
- CSUB (Conditional Subtract)
- CSUBN (Conditional Subtract Not)
- SEL (Select)
- SELN (Select Not)

The Conditional Instructions provide efficient alternatives to conditional jumps around very short sequences of code. All conditional instructions use a condition operand that controls the execution of the instruction. The condition operand is a data register, with any non-zero value interpreted as TRUE, and a zero value interpreted as FALSE.

Logical

The TriCore architecture provides a complete set of two-operand, bit-wise logic operations:

- AND
- OR
- XOR
- AND
- NOR
- XNOR

There are also negations of one of the inputs:

- ANDN
- ORN

Count Leading Zeroes, Ones, & Signs

Three Count Leading instructions provide efficient support for normalization of numerical results, prioritization, and certain graphics operations:

- CLZ (Count Leading Zeros)
- CLO (Count Leading Ones)
- CLS (Count Leading Signs)

These instructions determine the amount of left shifting necessary to remove redundant zeros, ones or signs.

The Count Leading instructions are useful for parsing certain Huffman codes and bit strings consisting of boolean flags, since the code or bit string can be quickly classified by determining the position of the first one (scanning from left to right).

Shift

The shift instructions support multi-bit left and right shifts. The shift amount is specified by a signed integer (n), which may be the contents of a register or a sign-extended constant in the instruction.

Bit-Field Extract and Insert

The TriCore architecture supports two bit-field extract instructions:

- EXTR
- EXTR.U

The EXTR.U and EXTR instructions extract w (width) consecutive bits from the source, beginning with the bit number specified by the pos (position) operand.

The width and position can be specified by two immediate values, by a data register and an immediate value, or by a data register pair.

8.4.2 DSP & Packed Arithmetic

DSP arithmetic instructions operate on 16-bit, signed fractional data in the 1.15 format (also known as Q15) and 32-bit signed fractional data in 1.31 format (also known as Q31).

Data values in this format have a single, high-order sign bit, with a value of 0 or -1, followed by an implied binary point and fraction. Their values are in the range [-1, 1).

16-bit DSP data is loaded into the most significant half of a data register, with the 16 least-significant bits set to zero. The left alignment of 16-bit data allows it to be directly added to 32-bit data in 1.31 format. All other fractional formats can be synthesized by explicitly shifting data as required.

Operations created for this format are multiplication, multiply-add and multiply-subtract. The signed fractional formats 1.15 and 1.31 are supported with the MUL.Q and MULR.Q instructions. These instructions operate on 2 left-justified, signed fractions, and return a 32-bit signed fraction.

Scaling

The multiplier result can be shifted in two ways:

- Left shifted by 1
  - 1 sign bit is suppressed and the result is left-aligned, thus conserving the input format.
- Not shifted
  - The result retains its 2 sign bits (2.30 format).
  - This format can be used with IIR filters, in which some of the coefficients are between 1 and 2, and to have 1 guard bit for accumulation.

Special case = -1 * -1 => +1

When multiplying the two maximum negative values (-1), the result should be the maximum positive number (+1). For example,

0x8000 * 0x8000 = 0x4000 0000 [8.1]

This is correctly interpreted in Q format as:

-1(1.15 format) * -1(1.15 format) = +1 (2.30 format) [8.2]

However, when the result is shifted left by 1, the result is 0x8000 0000, which is incorrectly interpreted as:

-1(1.15 format) * -1(1.15 format) = -1 (1.31 format) [8.3]

To avoid this problem the result of a Q format operation (-1 * -1) that has been left-shifted by 1 (left-justified), is saturated to the maximum positive value:

0x8000 * 0x8000 = 0x7FFF FFFF [8.4]

This is correctly interpreted in Q format as:

-1(1.15 format) * -1(1.15 format) = (nearest representation of)+1 (1.31 format) [8.5]

This operation is completely transparent to the user and does not set the overflow flags.

Guard bits

When accumulating sums (in filter calculations for example) guard bits are often required to prevent overflow. The instruction set directly supports the use of 1 guard bit when using a 32-bit accumulator. When more guard bits are required, a register pair (64 bits) can be used.

Rounding

Rounding is used to retain the 16-bit most-significant bits of a 32-bit result. Rounding is combined with the MUL, MADD and MSUB instructions, and is implemented by adding 1 to bit 15 of a 32-bit register.

Overflow & Saturation

Users can select normal or saturating instruction forms for multiply, multiply-add, and multiply-subtract. The normal forms set the overflow flag in the PSW (Processor Status Word (PSW) but do not produce a saturated result. The saturating forms produce

saturated values when overflow occurs. Variants are supported for both signed and unsigned operations.

Sticky Advance Overflow (SAV) & Block Scaling in FFT

The Sticky Advance Overflow (SAV) bit, which is set whenever an overflow “almost” occurred, can be used in block scaling of intermediate results during an FFT (Fast Fourier Transformation) calculation.

Before each pass of applying a butterfly operation, the SAV bit is cleared. After the pass the SAV bit is tested. If it is set then all of the data is scaled (using an arithmetic right shift) before starting the next pass. This procedure gives the greatest dynamic range for intermediate results without the risk of overflow.

8.4.3 Packed Arithmetic

The packed arithmetic instructions partition a 32-bit word into several identical objects which can then be fetched, stored and operated on in parallel. These instructions in particular, allow the full exploitation of the 32-bit word of the TriCore architecture in signal and data processing applications. The TriCore architecture supports two packed formats: Packed Halfword and Packed Byte.

Packed Halfword Data Format

This format (**Figure 8**) divides the 32-bit word into two, 16-bit (halfword) values. Instructions which operate on data in this way are denoted in the instruction mnemonic by the “.H” and “.HU” data type modifiers.

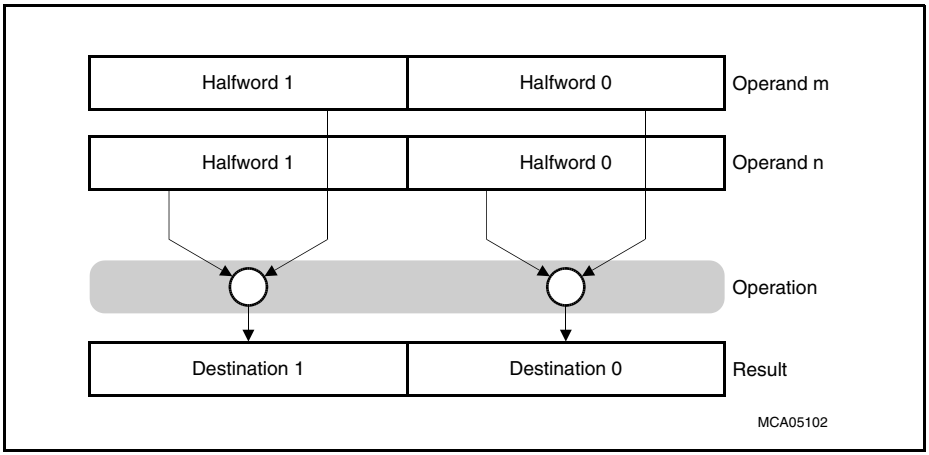


Figure 8 Packed Halfword Data Format

Packed Byte Data Format

This packed format (**Figure 9**) divides the 32-bit word into four 8-bit values. Instructions which operate on the data in this way are denoted by the ".B" and ".BU" data type modifiers.

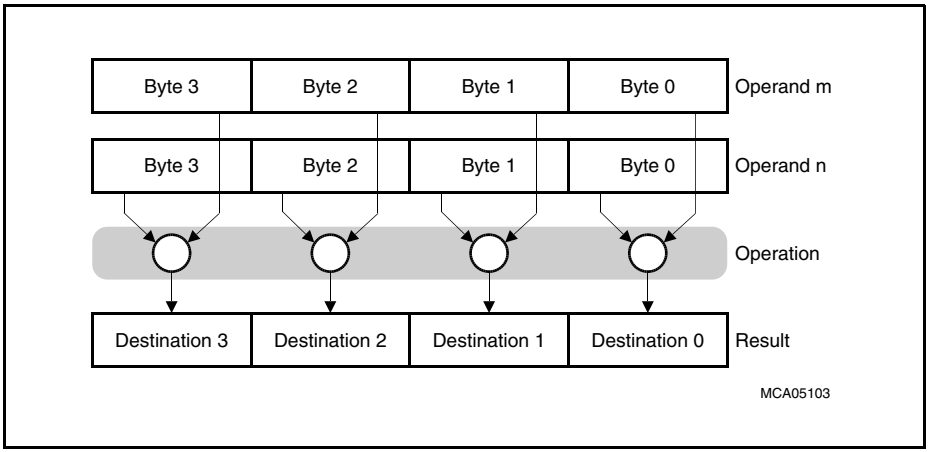


Figure 9 Packed Byte Data Format

The loading and storing of packed values into data registers is supported by the normal Load Word and Store Word instructions (LD.W and ST.W). The packed objects can then be manipulated in parallel by a set of special packed arithmetic instructions that perform arithmetic operations such as addition, subtraction and multiplication.

Addition is performed on individual packed bytes or halfwords using the ADD.B and ADD.H instructions and their saturating variations ADDS.B and ADDS.H.

ADD.B ignores overflow/underflow within individual bytes. ADDS.B will saturate individual bytes to the most positive, 8-bit signed integer (127) on individual overflow, or to the most negative, 8-bit signed integer (-128) on individual underflow.

Similarly, the ADD.H instruction ignores overflow/underflow within individual halfwords, while the ADDS.H will saturate individual halfwords to the most positive, 16-bit signed integer (2^{15}-1) on individual overflow, or to the most negative 16-bit signed integer (-2^{15}) on individual underflow.

Saturation for unsigned integers is also supported by the ADDS.BU and ADDS.HU instructions.

Besides addition, arithmetic on packed data includes subtraction, multiplication, absolute value and absolute difference.

8.5 Comparison Instructions

The compare (and conditional jump) instructions use a compare operation on the contents of two registers. The boolean result (1 = true and 0 = false) is stored in the least-significant bit of a data register. The remaining bits in the register are cleared to zero.

Figure 10 illustrates the operation of the LT (Less Than) compare instruction.

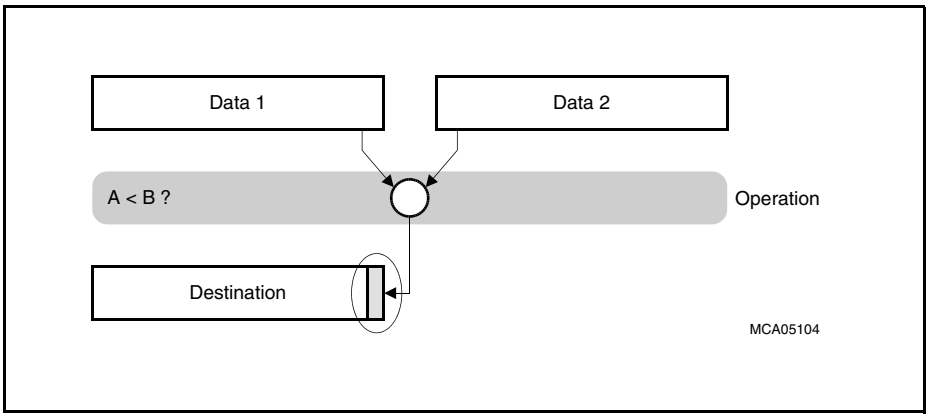


Figure 10 Less Than (LT) Comparison

8.6 Bit Operations

Some TriCore instructions operate on single bits. There are eight instructions for combinatorial logic functions with two inputs, and twelve instructions with three inputs.

8.6.1 Two-Input Boolean Operations

The one-bit result of a two-input function is stored in the least-significant bit of the destination data register and the most-significant 31 bits are set to zero (see **Figure 11**). The source bits can be any bit of any data register.

The available Boolean operations are:

– AND
– NAND
– OR
– NOR
– XOR
– XNOR
– ANDN
– ORN

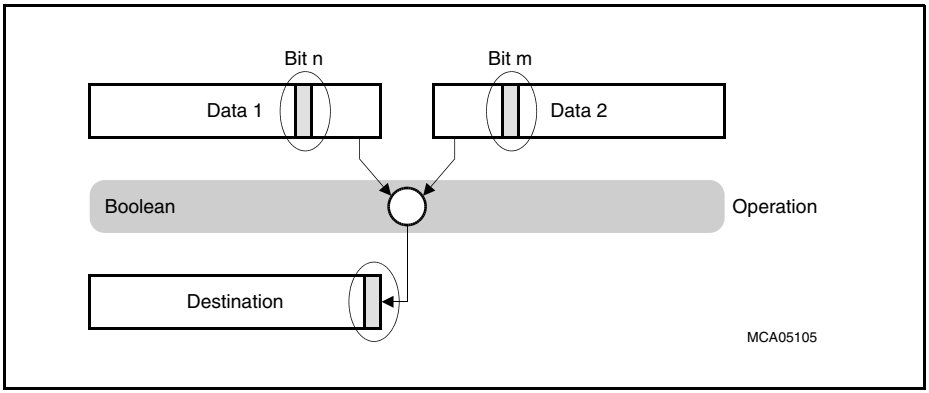


Figure 11 Two-Input Boolean Operations

8.6.2 Three-Input Boolean Operations

The three-input Boolean operations are used to evaluate complex Boolean operations where the output of a two-input instruction together with the least-significant bit of a third data register, forms the input to a further operation. The result is written to bit 0 of the third data register with the remaining bits unchanged (**Figure 12**).

The available Boolean operations are:

– AND.AND.T
– AND.ANDN.T
– AND.NOR.T
– AND.OR.T
– OR.AND.T
– OR.ANDN.T
– OR.NOR.T
– OR.OR.T

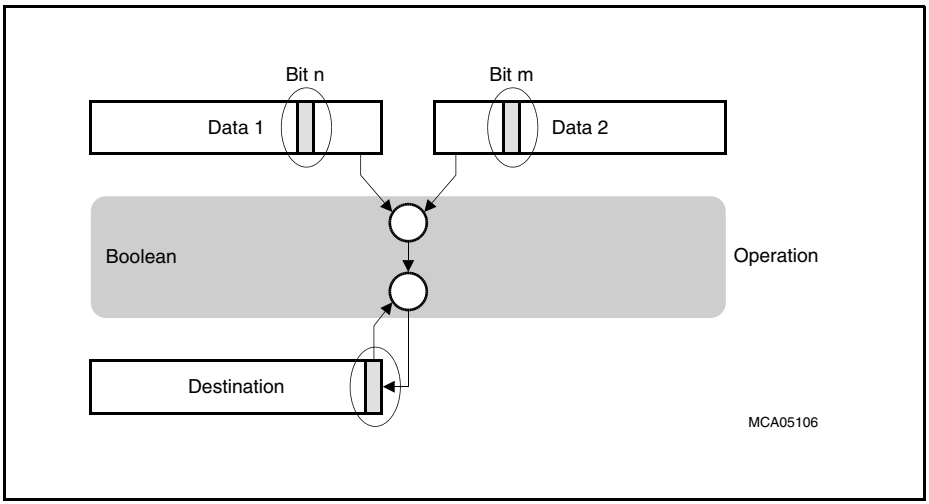


Figure 12 3-Input Boolean Operation

8.7 Address Arithmetic and Address Comparison

The TriCore architecture provides selected arithmetic operations on the address registers. These operations supplement the address calculations inherent in the addressing modes used by the load and store instructions.

As with the comparison instructions that use the data registers, the comparison instructions using the address registers put the result of the comparison in the least-significant bit of the destination data register and clear the remaining register bits to zeros. An example using the Less Than (LT.A) instruction is shown here in **Figure 13**.

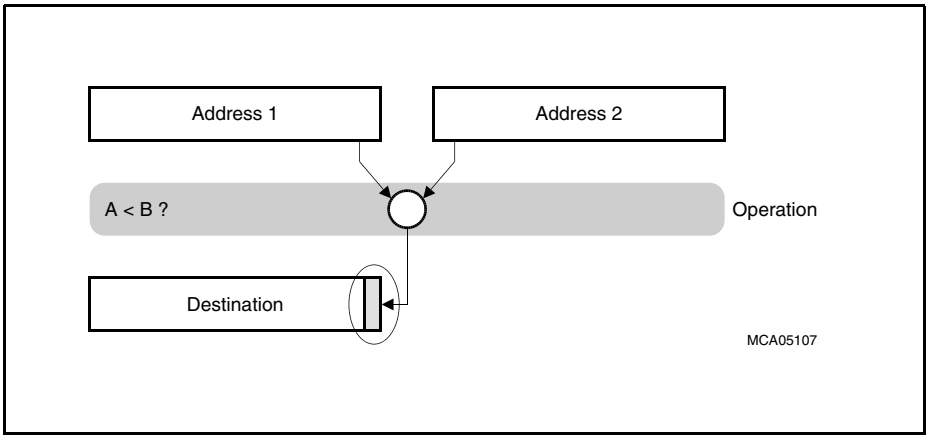


Figure 13 LT.A Comparison Operation

8.8 Branch Instructions

Branch instructions change the flow of program control by modifying the value in the PC register. There are two types of branch instructions: conditional and unconditional. Whether or not a conditional branch is taken depends on the result of a Boolean compare operation, rather than on the state of condition codes.

8.9 System Instructions

System Instructions can access and control various system services, including interrupts and TriCore's debugging facilities. Some instructions can be executed only in supervisor mode, such as MTCR for example, to write to a control register. Other instructions can be executed in either supervisor or user mode. There are also instructions that read and write the PSW and PCXI registers for both user and supervisor-only mode programs. The Load/Store Upper/Lower Context instructions explicitly save and restore a task's upper and lower contexts.

8.10 16-bit Instructions

The 16-bit instructions are a subset of the 32-bit instruction set, chosen because of their frequency of use. They significantly reduce static code size and thus reduce the cost of code memory and provide a higher effective instruction bandwidth. Because the 16-bit and 32-bit instructions all differ in the primary opcode, the two instruction sizes can be freely intermixed.

The 16-bit instructions are formed by imposing one or more of the following format constraints:

- Smaller constants
- Smaller displacements
- Smaller offsets
- Implicit source, destination or base address registers
- Combined source and destination registers (the two-operand format)

The 16-bit load and store instructions support only a limited set of addressing modes.

9 TriCore-1.3 Architecture Summary

The TriCore-1.3 core implements a Harvard architecture with separate address and data buses for program and data memories. Instruction fetches can be handled in parallel with data accesses. The superscalar core consists of two major pipelines with four stages each, and one minor pipeline for loop control. The three pipelines operate in parallel, allowing up to three instructions to execute in one cycle.

The core is a RISC Load/Store machine. All arithmetic instructions use registers. There are two General-Purpose Register Files; one comprised of 16 address registers and the other comprised of 16 data registers. The TriCore Instruction Set Architecture (ISA) provides a set of Load/Store instructions that fetch the data from the memory and store it back to the memories. This configuration allows fast interrupt response.

The TriCore-1.3 core's Integer Execute Unit consists of a dual Multiply Accumulate Module (MAC), an Arithmetic and Logic Unit (ALU), and a small tightly coupled Co-processor interface with access to the Register File. The TriCore-1.3 core can process two 16 x 16 Multiply-Accumulates per clock cycle.

The Flexible Peripheral Interconnect Bus (FPI Bus) easily connects the core to memory, internal and external peripherals, or other CPUs for example. The internal memory interfaces (both data and instruction) are both connected to the LMB Bus through individual interfaces. The Scratchpad RAMs (SPRs) ensure the timing of critical routines without having to rely on the caches.

The minimum TriCore implementation consists of a CPU core. However, depending on individual design requirements, peripheral and memory modules can be added to the core design from Infineon's own or another preferred library. These modules normally connect via the FPI Bus (Flexible Peripheral Interconnect Bus). High-performance memory can also connect directly to the Local Memory Bus hub (LMBh).

The core interfaces to the FPI bus through the LFI bridge for easy interconnection to all kinds of internal and external peripherals, memories and different active bus agents, such as CPUs and DMA/PCP controllers. **Figure 14** shows the CPU core with optional data and instruction memories.

The following sections of this chapter discuss the core and the optional modules that can comprise a TriCore-1.3 SOC (System-On-a-Chip).

9.1 Program and Data Memories

TriCore implements a Harvard architecture with separate program and data memories. The Program and Data memory blocks contain Scratch Pad RAMs (SPR) and/or cache memory ($). On each side the combined maximum is 64k of memory.

The cache memory, on each side, is 2-way associative and can hold up to 16k bytes in 4k increments. The total amount of memory as well as the partition between SPR and cache, can be customised for individual requirements at build time.

9.2 TriCore Bus Interfaces

The minimum TriCore implementation consists of a CPU core. However, depending on individual design requirements, peripheral and memory modules can be added to the core design from Infineon's own or another preferred library. These modules normally connect via the FPI (Flexible Peripheral Interconnect Bus). High-performance memory can also connect directly to the Local Memory Bus hub (LMBh). The core interfaces to the FPI bus through the LFI bridge for easy interconnection to all kinds of internal and external peripherals, memories and different active bus agents, such as CPUs and DMA/ PCP controllers.

Figure 14 shows the CPU core with optional data and instruction memories.

The following sections of this chapter discuss the core and the optional modules that can comprise a TriCore 1.3 SOC (System-On-a-Chip).

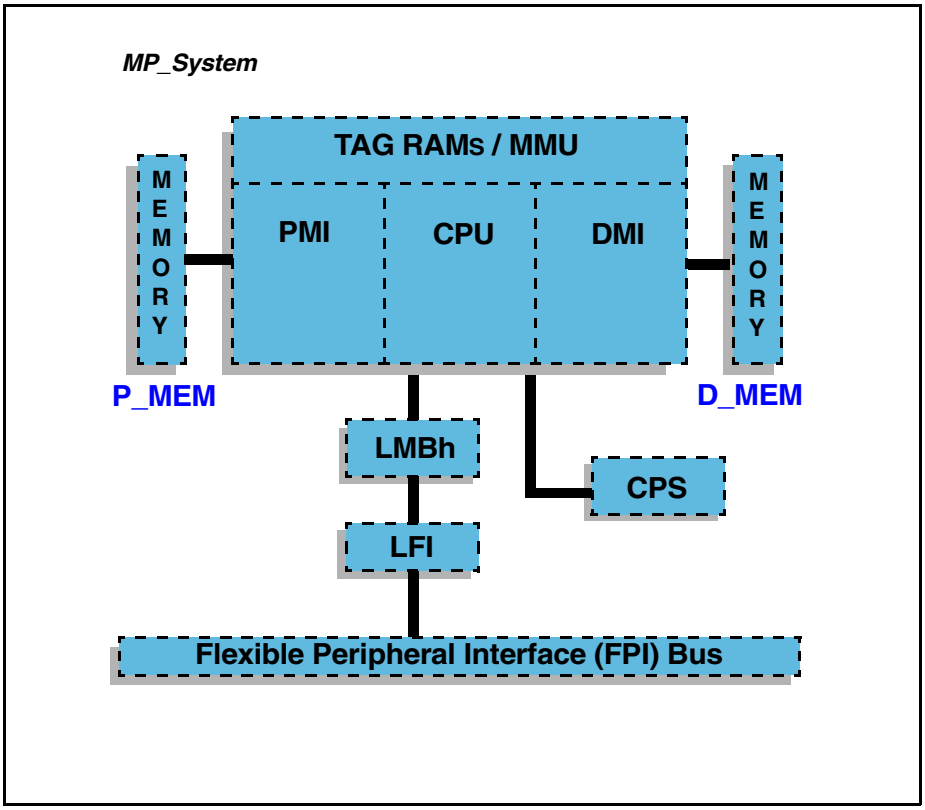


Figure 14 TriCore-1.3 Microprocessor Core

9.3 Local Memory Bus Hub (LMBh)

The LMBh is optimized for speed in order to support devices requiring a fast response time. Local memory (acting like level 2 cache) can be connected to this hub to give PMI and DMI fast access.

The Local Memory Bus (LMB) is a synchronous, pipelined, split bus with variable block size transfer support. Additional features include:

- 32-bit address, 64-bit data
- Central simple per-cycle arbitration
- Slave controlled wait state insertion
- Multi-master capability
- Burst mode read/write to memories.

The LMB can also be configured to use off-chip memories.

9.4 CPU Processor Slave (CPS)

The CPS contains two main blocks: the Interrupt Control Unit (ICU) and the Debug Trace unit. The ICU provides the interface between the CPU and the interrupt system. It can handle up to 255 interrupts per system. The Debug Trace unit allows real time tracing of program or data.

9.5 LMB to PFI Interface (LFI)

The LFI bridges the FPI to the LMB via FIFOs. These de-couple the transfer of data between the two busses, allowing each bus to operate at its own optimal rate.

LFI's features include:

- Burst/Single transactions, from FPI to LMB
- Burst/Single transactions from LMB to FPI
- Transactions are pipelined on each side of the bridge
- Programmable split LMB to FPI read transactions
- Supports all FPI data sizes
- Can handle abort, error and retry conditions on both sides of the bridge

9.6 FPI Bus Overview

The FPI Bus (Flexible Peripheral Interconnect Bus) is an on-chip bus designed to be used in modular, highly integrated system chips. The FPI Bus is designed for memory and I/O mapped data transfers between its bus agents, where bus agents are on-chip function blocks (modules) that are equipped with an FPI Bus interface. It is a de-multiplexed bus with up to 32 address bits and 64 data bits. Peak throughput is 800 Mbytes/s at 100 MHz.

There is no limit to the number of peripheral modules that can be connected to the FPI Bus. Additional features include:

- Multimaster capability (up to 16 masters)
- De-multiplexed operation
- Clock synchronous
- 8-/16-/32- and 64-bit data transfers
- Broad range of transfer types from single to multiple data transfers
- Flexible bus protocol, which can be tailored to your application needs

There are three types of agents possible on the FPI Bus (see **Figure 15**):

- Master agents which can initiate and control transactions
- Slave agents, which only support simple read and write of registers and are not actively operating on the bus protocol.
- Master-Slave Agents, which support advanced features like split read transfer support and error handling. Depending on the type of transaction these agents may act as master, slave or both.

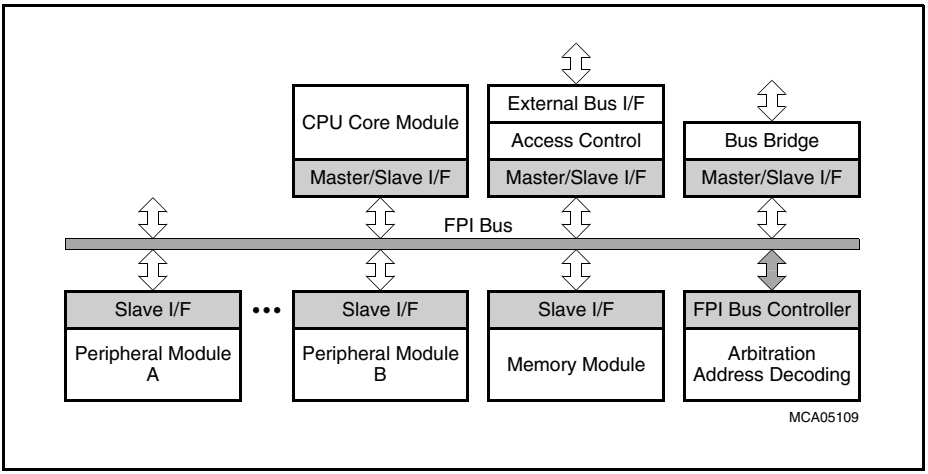


Figure 15 Examples of Modules within an FPI Bus-Based System

10 TriCore Software Development Tools

The TriCore architecture is well supported by a robust set of hardware and software development tools (**Figure 16**). These tools include the TriCore Instruction Set Simulator (TSIM), compiler-assembler debugger tool chain, real-time operating systems and emulators. The instruction set architecture was developed in close consultation with the third party providers of these tools and the TriCore Instruction Set Simulator (TSIM) is bundled together with complete (debugger-compiler-assembler-linker-loader) tool chains from several vendors.

Information on all of the Development Tools and vendors can be found at the SPACE program website: **www.spacetools.com**.

Evaluation kits (PC and UNIX versions) are available free of cost to qualified customers. System designers can not only perform price-performance trade-off's on this instruction accurate simulator, but can also begin their software development and debugging.

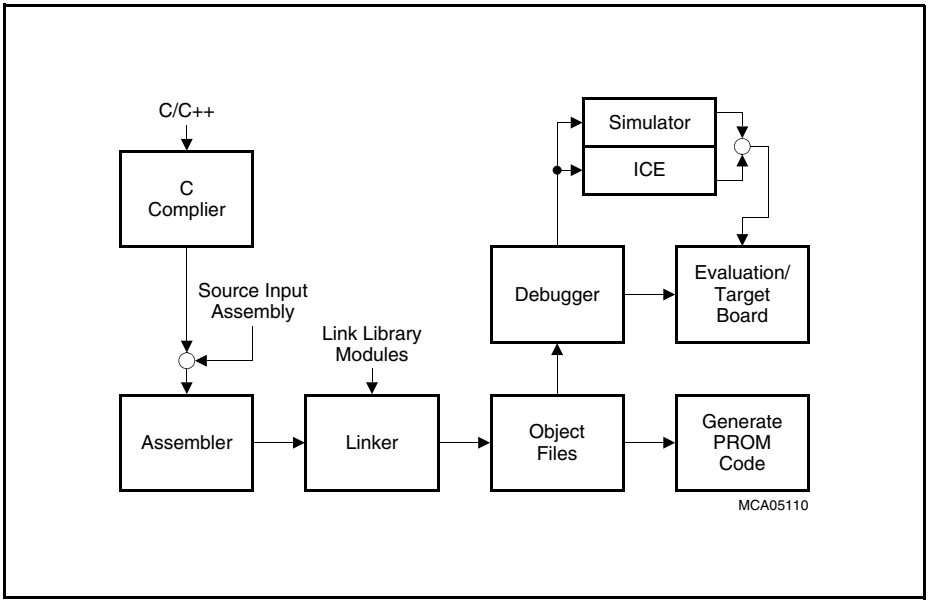


Figure 16 TriCore Development Tools

10.1 TSIM - TriCore Instruction Set Simulator

TSIM is a configurable, instruction-accurate model of the TriCore-1.3 core architecture that is integrated into all supported source-level debuggers. TSIM provides a simulation environment that models the TriCore core, memory configuration and interrupt mechanism. TSIM is useful for performance and trade-off analysis, and for developing and debugging a customized design.

The TriCore-1.3 core can be reprogrammed to evaluate an implementation approach by changing the memory parameters in the TSIM memory configuration file (**MConfig**). Interrupt events can be specified in the TSIM interrupt configuration file (**IConfig**) to evaluate interrupt operation and performance.

The TSIM peripheral configuration file (**PConfig**) tells your program how to communicate with the external peripherals used in a given implementation.

Figure 17 shows an overview of the simulation environment.

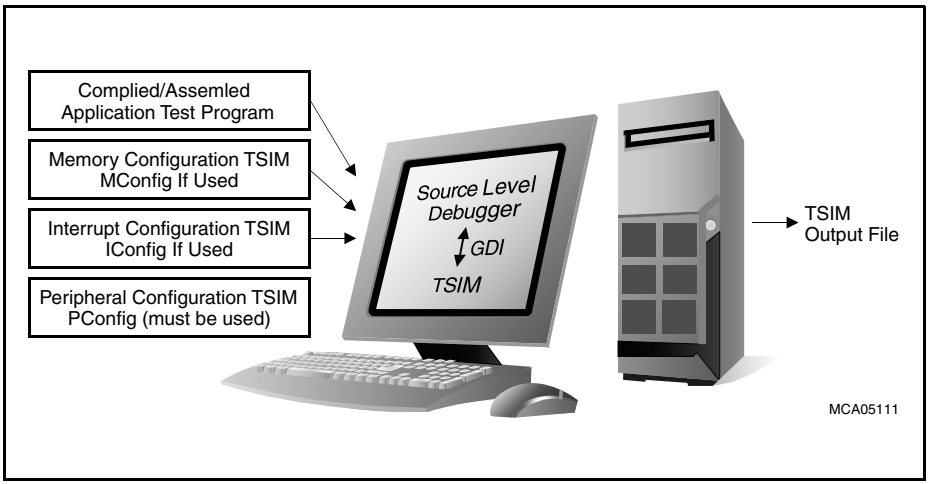


Figure 17 TSIM Simulation Environment

Please refer to Infineon's TriCore Instruction Set Simulator User's Guide for more information.

11 DSP Example

The TriCore-1.3 superscalar architecture consists of three units - the Integer Execution Unit, the Load/Store Unit and the Loop Unit, allowing the issue of up to three instructions per clock cycle. **Figure 18** shows the different possible instruction issue combinations.

The highest issue rate is achieved when a load/store, integer and loop instruction are all available. This issue rate is easy to reach during the inner loop of many DSP routines, allowing TriCore to deliver a sustained DSP throughput of 2 16x16 MACs per clock. The following is an example of how this works:

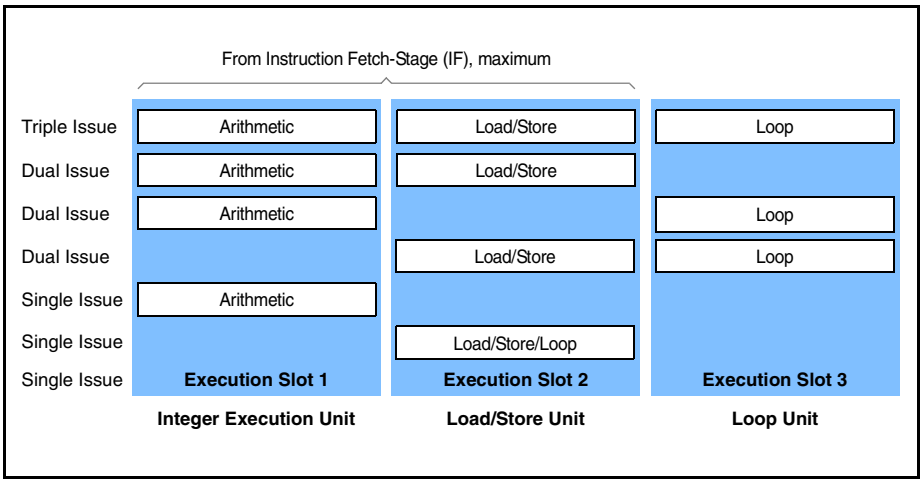


Figure 18 Superscalar Instruction Issue

This superscalar implementation can process two 16x16 Multiply-Accumulates per clock cycle. Assume for example, that the following equation needs to be calculated:

$$\sum_{i=0}^{n} c_i x_i = c_0 x_0 + c_1 x_1 + \ldots + c_n$$

If n=255 (as in a 256-tap filter), the table below summarizes the execution unit utilization, assuming 16-bit fixed point data. In this example, eight 16x16 MACs are calculated for each loop iteration:

Clock	Integer Unit	Load/Store Unit	Loop Unit
clock 1	-	Load C_0, C_1, C_2, C_3	-
clock 2	-	Load X_0, X_1, X_2, X_3	-
clock 3	MAC C_0X_0, MAC C_1X_1	Load C_4, C_5, C_6, C_7	Loop Start
clock 4	MAC C_2X_2,, MAC C_3X_3	Load X_4, X_5, X_6, X_7	-
clock 5	MAC C_4X_4, MAC C_5X_5	Load C_8, C_9, C_{10}, C_{11}	-
clock 6	MAC C_6X_6,,MAC C_7X_7	Load X_8, X_9, X_{10}, X_{11}	Loop
...	...	...	-
clock 130	MAC $C_{254}X_{254}$, MAC $C_{255}X_{255}$	-	-
clock 131	-	Store Result	-

In this example, 16-bit operands are moved four-at-a-time into two 32-bit registers using 64-bit load operations. Eight operands are moved into four registers, then two dual-MAC operations process them.

In parallel with this processing, the next 8 operands are moved into four other registers. These other registers are then used in the next two MAC operations. While the next two MACs are being performed, the first set of registers is loaded with the next 8 operands. The loads and MACs are therefore interleaved, with loads "ping-ponging" between two sets of registers. Sustained dual-MAC DSP throughput is therefore obtained.

12 Glossary

Reference	Definition
API	Application Program Interface - A set of routines, protocols and tools for building software applications
ASIC	Application Specific Integrated Circuit
Ax	Address Registers
BCU	Bus Control Unit
BIST	Built-In Self Test
BIV	Base Address of Interrupt Vector Table
BPI	Bus Peripheral Interface
BTV	Base Address of Trap Vector Table
Context	Every Task has a Context. The Context is everything the processor needs in order to define the state of the associated task and enable its continued execution. Context's are subdivided into Upper & Lower Contexts. Upper consists of upper Address (A10 - A15) and Data (D8 - D15) Registers and Lower is the lower Address (A2 - A7) and Data (D0 - D7) Data Registers.
CPM	Code Protection Mode
CPR	Code Segment Protection Register
CPS	CPU Slave
CPU	Central Processing Unit
CREVT	Emulator Resource Protection Event Specifier Register
CSA	Context Save Area - Each CSA can hold 1 upper or 1 lower Context (see Context). CSAs are linked together through a Link Word
CSFR	Core Special Function Register (SFR)
CVE	Core Verification Environment
DBGSR	Debug Status Register
DCU	Data Control Unit
DCX	Debug Context
DFF	D-type Flip Flop
DMA	Direct Memory Access - A technique for transferring data directly between two peripherals (usually memory and an I/O device) with only minimal intervention by the processor. DMA transfers are managed by a third peripheral called a DMA controller

Reference	Definition
DMS	Debug Monitor Start Address
DMU	Data Memory Unit
DPM	Data Protection Mode
DPR	Data Segment Protection Register
DRAM	Direct Random Access Memory
DSP	Digital Signal Processing
Dx	Data Registers
EBU	External Bus Unit
EMU	Emulation Monitoring Unit
EXEVT	External Break Input Event Specifier Register
FFT	Fast Fourier Transformations
FLASH	also known as ‘Flash RAM’ - constantly powered, non-volatile memory that can be erased and re-programmed.
FPI	Flexible Peripheral Interface - Used for on-chip interconnections, connecting the core with peripherals including ports and the External Bus Controller.
FPU	Floating Point Unit
GDI	Graphical Device Interface
GPR	General Purpose Register
HDL	Hardware Definition Language - used in engineering to describe the design and testbench of a system. HDL Software simulators are then used to verify the Design with the testbench. See also VHDL
HLL	High Level Language
IC	Integrated Circuit
ICR	Interrupt Unit Control Register
ICU	Interrupt Control Unit
ID	Identity / Identification
IEEE	Institute of Electrical & Electronics Engineers
I/F	Interface
IIR	Infinite Impulse Response A digital filter with internal registers that hold past output of the filter
ISA	Instruction Set Architecture
ISP	Interrupt Stack Pointer

Reference	Definition
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LCU	LMB Control Unit
LDCUX	Instruction to Load Upper Context from memory
LDLCX	Instruction to Load Lower Context from memory
LEADDR	LMB Error Address
LEATT	LMB Error Attributes
LEDAT	LMB Error Data
LFI	LMB to FPI Interface: A bridge between the two main buses, LMB and FPI
LMB	Local Memory Bus - a high speed, high bandwidth local memory bus supporting devices with fast response times, targeted at providing the local CPU with fast access to local memory
LMBH	Local Memory Bus (LMB) Hub
LMU	Local Memory Unit
MAC	Multiply and Accumulate
MCU	MicroController Unit
MIPS	Million Instructions Per Second
MMU	Memory Management Unit - translates virtual addresses issued by the load / store, instruction fetch unit into physical addresses to feed into the PMU and DMU respectively.
MP	MicroProcessor
MTCR	Move To Control Register
OCDS	On-Chip Debug Support
OTP	One-Time Programmable
PDA	Personal Digital Assistant
PC	Program Counter
PCP	Peripheral Control Processor - an I/O control processor that performs tasks typically handled by a dedicated DMA controller and CPU interrupt service routines. The PCP off-loads the CPU from time-critical interrupts, easing implementation of systems based on Operating Systems. PCP is optimized to efficiently support DMA-type bus transactions to and from arbitrary devices and memory addresses.
PCXI	Previous Context Information Register

Reference	Definition
PMU	Program Memory Unit
PRS	Protection Register Set
PSW	Processor Status Word
PTE	Page Table Entry
RAM	Random Access Memory
RFE	Return From Exception
RISC	Reduced Instruction Set Computer. Describes the architecture of a processor family.
ROM	Read Only Memory
RSLCX	Restore Lower Context instruction
RSTV	Reset Overflow Bits
RTL	Register Transfer Level - Describes a System in terms of registers, combinational circuitry, low-level buses and control circuits. It is used for developing and testing internal architecture and control logic within an IC component, so that the design satisfies the required functionality and timing constraints of the IC. (source - RASSP Taxonomy Working Group)
RTOS	Real-Time Operating System
SAV	Sticky Advance Overflow
SFR	Special Function Register
SIMD	Single Instruction Multiple Data
SMT	Software Managed Tasks
SOC	System-On-a-Chip
SP	Stack Pointer
SPR	Scratch Pad RAM
SRAM	Static Random Access Memory
SRN	Service Request Node
SRPN	Service Request Priority Number
STLCX	Store Lower Context instruction
STPG	Statis Test Pattern Generation
STUCX	Store Upper Context
SVLCX	Save Lower Context instruction
SYSCON	System Control Register

Reference	Definition
Task	Refers to an independent thread of control. There are two types of tasks: Software Managed Tasks (SMTs) and Interrupt Service Routines (ISRs)
TC	Abbreviation for TriCore (i.e. TC2 - TriCore v2.0)
TFA	Translation Fault Address
TIN	Trap Identification Number - Identifies the cause of a trap within its class (TriCore has 8 Trap classes).
TLB	Translation Lookaside Buffer
TOS	Type Of Service
TPA	Translation Physical Address
TPX	Translation Page Index
TSIM	TriCore Instruction Set Simulator (Tricore SIMulator) - A configurable, instruction-accurate model of the TriCore core architecture, providing a simulation environment that models the core, memory configuration and interrupt mechanism. Used for debugging and developing customized designs
VHDL	Very High Definition Language
VHDL	VHSIC Hardware Description Language - The standard for the design and description of electronic systems

Infineon goes for Business Excellence

“Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.
Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction.”

Dr. Ulrich Schumacher

http://www.infineon.com

Published by Infineon Technologies AG