Xtensa® LX Microprocessor

Overview Handbook

A Summary of the Xtensa® LX Microprocessor Data Book

For Xtensa® LX Processor Cores

Tensilica, Inc.
3255-6 Scott Blvd
Santa Clara, CA 95054
(408) 986-8000
fax (408) 986-8919
www.tensilica.com

Issue Date: 9/2004

PD-04-2508-10-00

Tensilica, Inc.
3255-6 Scott Blvd
Santa Clara, CA 95054
(408) 986-8000
fax (408) 986-8919
www.tensilica.com

Contents

List of Figures

List of Tables

Preface

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- $ represents your shell prompt, in user-session examples.
- **`literal_input`** indicates literal command-line input.
- *`variable`* indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *`... output ...`* indicates unspecified program output.
- *`[optional-variable]`* indicates an optional parameter.
- **`[`***`variable`***`]`** indicates a parameter within literal square-braces.
- **`{`***`variable`***`}`** indicates a parameter within literal curly-braces.
- **`(`***`variable`***`)`** indicates a parameter within literal parentheses.
- | means *OR*.
- *`(var1 | var2)`* indicates a required choice between one of multiple parameters.
- *`[var1 | var2]`* indicates an optional choice between one of multiple parameters.
- *`var1 [, varn]*`* indicates a list of 1 or more parameters (0 or more repetitions).
- `4'b0010` is a 4-bit value specified in binary.
- `12'o7016` is a 12-bit value specified in octal.
- `10'd4839` is a 10-bit value specified in decimal.
- `32'hff2a` or `32'HFF2A` is a 32-bit value specified in hexadecimal.

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.
- *flush* is deprecated due to potential ambiguity (it may mean *write-back* or *discard)*.
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- *word* means 4 bytes.

Related Tensilica Documents

- *GNU Assembler User's Guide*
- *GNU Binary Utilities User's Guide*
- *GNU Debugger User's Guide*
- *GNU Linker User's Guide*
- *GNU Profiler User's Guide*
- *On-Chip Debugging Guide*
- *Red Hat newlib C Library Reference Manual*
- *Red Hat newlib C Math Library Reference Manual*
- *Tensilica Instruction Extension (TIE) Language Reference Manual*
- *Tensilica Instruction Extension (TIE) Language User's Guide*
- *Vectra™ LX DSP Engine Guide*
- *Xtensa® Bus Designer's Toolkit Guide*
- *Xtensa® Bus Interface User's Guide*
- *Xtensa® C and C++ Compiler User's Guide*
- *Xtensa ® LX Co-Simulation Model for Mentor Graphics Seamless® User's Guide*
- *Xtensa® Development Tools Installation Guide*
- *Xtensa® LX Hardware Implementation and Verification Guide*
- *Xtensa® Instruction Set Architecture (ISA) Reference Manual*
- *Xtensa® Instruction Set Simulator (ISS) User's Guide*
- *Xtensa® Linker Support Packages (LSPs) Reference Manual*
- *Xtensa® LX Microprocessor Data Book*
- *Xtensa® Microprocessor Emulation Kit–XT2000-X User's Guide*
- *Xtensa® LX Microprocessor Overview Handbook*
- *Xtensa® Microprocessor Programmer's Guide*
- *Xtensa® OSKit™ Guide*
- *Xtensa® Processor Extensions Synthesis (XPRES™) Compiler User's Guide*
- *Xtensa® Processor Interface Protocol Reference Manual*
- *Xtensa® Software Development Toolkit User's Guide*
- *Xtensa® System Software Reference Manual*

Introducing the Xtensa LX Processor Generator

Introduction to Configurable and Extensible Processor Cores

Tensilica's Xtensa technology provides SOC (system-on-chip) designers with the world's first configurable and extensible processor core supported by automatic hardware and software generation. The Xtensa line is the first microprocessor core family designed specifically to meet the wide range of performance requirements in today's SOC designs. Xtensa processors allow SOC designers to apply programmable processing power to multiple SOC application challenges through individually tailored processor implementations. Because it is unlike a conventional embedded processor core, Xtensa processors change the rules of the SOC game.

A few characteristics of typical deep-sub-micron IC design illustrate the challenge facing SOC design teams:

In a typical 0.13-micron standard-cell foundry process, silicon density routinely exceeds 100K usable gates per mm^2. Consequently, a low-cost chip (a 50mm^2 die) can carry 5M gates of logic. Simply because it's possible, a system designer somewhere will find ways to exploit this potential in any given market. The designer faces a set of daunting challenges, however:

- Design effort: In the past, silicon capacity and design-automation tools limited the practical size of an RTL block to no more than 100K gates. Improved logic-synthesis, place-and-route, and verification tools are raising that ceiling. Blocks of 500K gates are now within the capacity of these tools, but existing design methods are not keeping pace with silicon fabrication capacity, which now puts millions of gates on an SOC.
- Verification difficulty: The internal complexity of a typical logic block—hence the number of potential bugs—grows much more rapidly than does its gate count. System complexity increases much more rapidly than the number of constituent blocks. Similarly, verification complexity has grown disproportionately. Many teams that have recently developed real-world designs report that they now spend as much as 90% of their development effort on block- or system-level verification.
- Cost of fixing bugs: The cost of a design bug is going up. Much is made of the rising cost of deep-sub-micron IC masks—the cost of a full mask set is approaching $1M. However, mask costs are just the tip of the iceberg. The larger teams required by complex SOC designs, higher staff costs, bigger NRE fees, and lost profitability and market share make show-stopper design bugs intolerable. Design methods that reduce the occurrence of—or permit painless workarounds for—such show-stoppers pay for themselves rapidly.

- Late hardware/software integration: All embedded systems now contain significant amounts of software or firmware. Software integration is typically the last step in the system-development process and routinely bears the blame for program delays. Late hardware/software validation is widely viewed as a critical risk for new product-development projects.
- Complexity and changes in standards: Standard communication protocols are growing rapidly in complexity. The need to conserve scarce communications spectrum plus the inventiveness of modern protocol designers has resulted in complex new protocol standards (for example: IPv6 Internet Protocol packet forwarding, G.729 voice coding, JPEG2000 image compression, MPEG4 video, Rjindael AES encryption). These new standards demand much greater computational throughput than their predecessors because they are far more complex.

Although general-purpose embedded processors can handle many tasks, they often lack the bandwidth needed to perform complex, high-bandwidth data-processing tasks such as network packet processing, video processing, and digital cryptography. Chip designers have long turned to hardwired logic (blocks of RTL) to implement these key functions. As the complexity and bandwidth requirements of electronic systems increase, the total amount of on-chip logic rises steadily.

Even as designers wrestle with the growing resource demands of advanced chip design, they face two additional worries:

1. How do design teams ensure that the SOC specification really satisfies customer needs?
2. How do design teams ensure that the SOC's design really meets those specifications?

Further, a good design team will anticipate future needs of current customers and potential future customers—they have a built-in road map.

If the design team fails on the first criterion, the SOC may work perfectly, but it may have inadequate sales to justify the design and manufacturing effort. Changes in requirements may be driven by demands of specific key customers, or they may reflect overall market trends such as the emergence of new data formats, new protocol standards, or new feature expectations across an entire product category. While most SOC designs include some form of embedded control processor, the limited performance of these fixed-ISA (instruction-set architecture) processors often precludes them from being used for essential data processing tasks. Consequently, RTL blocks are designed to handle these high-performance tasks and therefore, software usually cannot be used to add or change fundamental new features after the SOC is designed and fabricated.

If the design team fails on the second criterion, additional time and resources must go towards changing or fixing the design. This resource drain delays market entry and causes companies to miss key customer commitments. Failure is most often realized as a program delay. This delay may come in the form of missed integration or verification milestones, or it may come in the form of hardware bugs—explicit logic errors that are not caught by the limited verification that is typical of hardware simulation. The underlying cause of failure might a subtle error in a single design element or it might be a miscommunication of requirements—subtle differences in assumptions between hardware and software teams, between design and verification teams, or between the SOC designers and the SOC library or foundry supplier. In any case, the design team may often be forced to jump into an urgent cycle of re-design, re-verification, and re-fabrication (respinning) of the chip. These design "spins" rarely take less than less than six months, which significantly disrupts product and business plans.

Using Microprocessor Cores in SOC Design

System developers are trying to solve two closely related problems. One problem is the need to significantly reduce the resource levels required to develop systems by making it easier to design the chips in those systems. The second problem is the need to make SOCs sufficiently flexible so that every new system design doesn't require a new SOC design.

The way to solve these two problems is to make the SOC sufficiently flexible so that one chip design will efficiently serve 10, 100, or 1000 different system designs while giving up none or perhaps just a few of the benefits of SOC integration. Solving these problems means creating chips that can satisfy the requirements of current system designs and next-generation designs, which amortizes SOC-development costs over a larger number of systems.

The specialized nature of individual embedded applications creates two issues for general-purpose processors in data-intensive embedded applications. First, there is a poor match between the critical functions needed by many embedded applications (for example, image, audio, and protocol processing) and a fixed-ISA processor's basic integer instruction set and register file. As a result of this mismatch, these critical embedded applications require an excessive number of computation cycles when run on general-purpose processors. In other words, general-purpose processors are too slow for many of these critical tasks. Second, narrowly focused, low-cost embedded devices (such as music players, printers, digital cameras, or electronic games) cannot take full advantage of a general-purpose processor's broad capabilities. Consequently, expensive silicon resources built into the processor are wasted in these applications because they're not needed by the specific embedded tasks assigned to the processor.

Many embedded systems interact closely with the real world or communicate complex data (like compressed video) at high rates. These data-intensive tasks could be performed by some hypothetical general-purpose microprocessor running at sky-high clock rates. For many embedded tasks, however, there’s no processor fast enough to serve as a practical alternative.

Even if there are processors with sufficiently high clock rates, the fastest available processors are typically too expensive and dissipate too much power (by orders of magnitude) to meet embedded-system design goals. Consequently, designers have traditionally turned to hard-wired circuits to perform these data-intensive functions.

In the past ten years, wide availability of logic synthesis and ASIC design tools has made RTL design the standard for hardware developers. RTL-based design is reasonably efficient (compared to custom, transistor-level circuit design) and can effectively exploit the intrinsic parallelism of many data-intensive problems. RTL design methods can often achieve tens or hundreds of times the performance achieved by a general-purpose processor running application firmware, often at lower power.

Like RTL-based design using logic synthesis, extensible-processor technology enables the design of high-speed logic blocks tailored to the assigned task. The key difference is that RTL state machines are realized in hardware and logic blocks based on extensible processors realize their state machines with firmware, which is better at dealing with complex algorithms and much easier to change if market requirements or protocol standards change.

Benefitting from Microprocessor Extensibility

A fully featured configurable and extensible processor consists of a base or foundation processor design and a design-tool environment that permits significant adaptation of that base processor by allowing system designers to change major processor functions, thus tuning the processor for specific applications. Typical forms of configurability include additions, deletions, and modifications to memories, to external-bus widths and handshake protocols, and to commonly used processor peripherals. An important superset of configurable processors is the extensible processor—a processor whose functions, especially its instruction set, can be extended by the application developer to include features never considered or imagined by designers of the original processor.

Changing the processor's instruction set, memories, and interfaces can make a significant difference in its efficiency and performance, particularly for the data-intensive applications that represent the "heavy lifting" of many embedded systems. These features might be too narrowly used to justify inclusion in a general-purpose instruction set and hand-designed processor hardware. All general-purpose processors are compromise designs, where features that provide modest benefits to all customers supersede features that provide dramatic benefits to a few. Design compromises are necessary because the historic costs and the difficulty of manual processor design mandate that only a few different processor designs can be built. Automatic processor generation dramatically reduces processor-development cost and development time so that inclusion of application-specific features to meet performance goals and deletion of unused features to meet cost goals suddenly becomes a very attractive alternative for system designers.

A configurable processor is a processor whose features can be pruned or augmented by parametric selection. Configurable processors may be implemented in many different hardware forms, ranging from ASICs with hardware implementation times of many weeks to FPGAs with implementation times of minutes. Extensible processors constitute an important superset of configurable processors. Extensible processors can be extended by the application developer to include instructions, execution units, registers, and other features that were never considered by the original processor designers.

For both configurable and extensible processors, the usefulness of configurablity and extensibility is strongly tied to the automatic availability of both hardware implementation and software environment supporting all configurability and extensibility aspects. Automated software support for extended features is especially important. Configuration or extension of the hardware without complementary enhancement of the compiler, assembler, simulator, debugger, real-time operating systems, and other software-development tools essentially leave the promises of performance and flexibility unfulfilled because the newly created processor can’t be programmed.

Extensibility's goal is to allow features to be added or adapted in any form that optimizes the processor’s cost, power, and application performance. In practice, the configurable and extensible features break down into four categories, as shown in Table 1.

Table 1. Configurable and Extensible Processor Features

Instruction Set	Memory System	Interface	Processor Peripherals
▪ Extensions to ALU functions using general registers (e.g. population count instruction) ▪ Coprocessors supporting application-specific data types (e.g. network packets, pixel blocks), including new registers and register files ▪ High-performance arithmetic and DSP (e.g. compound DSP instructions, vector/SIMD, floating point), often with wide execution units and registers ▪ Selection among function unit implementations (e.g. small iterative multiplier vs. pipelined array multiplier)	▪ Instruction-cache size, associativity, and line size ▪ Data-cache size, associativity, line size, and write policy ▪ Memory protection and translation (by segment, by page) ▪ Instruction and data RAM/ROM size and address range	▪ External bus interface width, protocol, and address decoding ▪ Direct connection of system control registers to internal registers and data ports ▪ Mapping of special-purpose memories (queues, multi-ported memories) into the address space of the processor ▪ TIE queues and ports ▪ State-visibility trace ports and JTAG-based debug ports	▪ Timers ▪ Interrupt controller: number, priority, type, fast switching registers ▪ Exception vectors addresses ▪ Remote debug and breakpoint controls

A block diagram for Tensilica's configurable, extensible Xtensa LX processor appears in Figure 1. The figure identifies baseline instruction-set architecture features, scalable register files, memories and interfaces, optional and configurable processor peripherals, selectable floating-point and DSP coprocessors, and facilities to integrate designer-defined instruction set extensions.

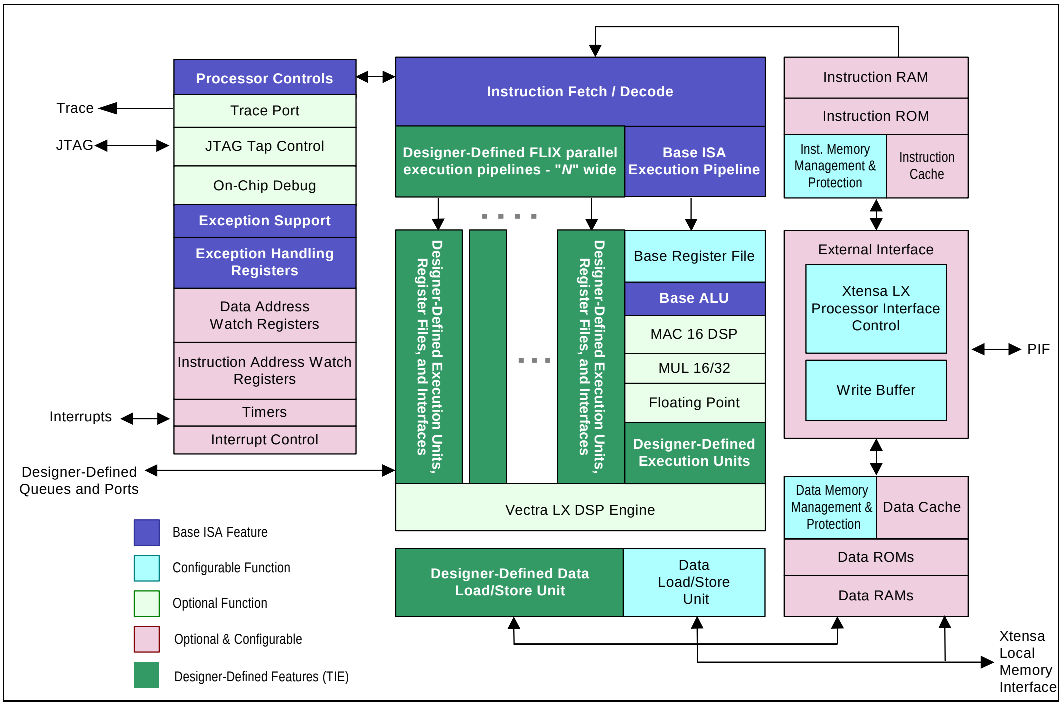

Processor Controls
Trace Port
JTAG Tap Control
On-Chip Debug
Exception Support
Exception Handling Registers
Data Address Watch Registers
Instruction Address Watch Registers
Timers
Interrupt Control
Trace
JTAG
Interrupts
Designer-Defined Queues and Ports
Instruction Fetch / Decode
Designer-Defined FLIX parallel execution pipelines - "N" wide
Base ISA Execution Pipeline
Designer-Defined Execution Units, Register Files, and Interfaces
Designer-Defined Execution Units, Register Files, and Interfaces
Base Register File
Base ALU
MAC 16 DSP
MUL 16/32
Floating Point
Designer-Defined Execution Units
Vectra LX DSP Engine
Designer-Defined Data Load/Store Unit
Data Load/Store Unit
Instruction RAM
Instruction ROM
Inst. Memory Management & Protection
Instruction Cache
External Interface
Xtensa LX Processor Interface Control
Write Buffer
PIF
Data Memory Management & Protection
Data Cache
Data ROMs
Data RAMs
Xtensa Local Memory Interface
Base ISA Feature
Configurable Function
Optional Function
Optional & Configurable
Designer-Defined Features (TIE)

Figure 1. Xtensa LX Processor Architectural Block Diagram

Processor extensibility serves as a particularly potent form of configurability because it handles a much wider range of applications and is easily usable by designers with a wide range of skills. Processor extensibility allows a system designer or application expert to directly incorporate proprietary insight about the application's functional and performance needs directly into instruction-set and register extensions.

How the Use of Microprocessor Cores for SOC Design Differs in Board-Level Designs

Hardwired RTL design has many attractive characteristics: small area, low power, and high throughput. With the advent of megagate SOCs, RTL's liabilities—difficult design, slow verification, and poor scalability for increasingly complex problems—are starting to dominate. A design methodology that retains most of RTL's benefits while reducing design time and risk has a natural appeal. The use of application-specific processors as an alternative to complex RTL design fits this need.

Application-specific processors can implement data-path operations that closely match those of RTL functions. The functional equivalents of RTL logic blocks are implemented using application-specific processors by adding execution units to the processor’s existing integer pipeline, additional registers and register files to the processor’s state, additional I/O ports, and other functions as needed by the specific application.

The Tensilica Instruction Extension (TIE) language, which resembles Verilog, is optimized for high-level specification of data-path functions in the form of instruction semantics and encoding. A TIE description is much more concise than RTL because it omits all sequential logic (including state machine descriptions), pipeline registers, and initialization sequences. The custom, designer-defined processor instructions and registers described in TIE become part of the processor’s programming model. These are therefore available to the firmware programmer using the same compiler and assembler that use the processor's base instructions and register set.

All operational sequencing within the processor's data path is controlled by firmware through the processor's pre-existing instruction-fetch, decode, and execution mechanisms. The firmware can be written in a high-level language such as C or C++, and the compiler will exploit the new processor features to accelerate the code execution speed.

Extended processors used as alternatives to RTL blocks routinely use the same structures as traditional RTL blocks: deep pipelines, parallel execution units, task-specific state registers, and wide data paths to local and global memories. These extended processors sustain the same high computational throughput and support the same low-level data interfaces as the RTL designs.

Control of the extended-processor data paths is very different, however. Cycle-by-cycle control of the processor's data paths is not embodied in hardwired state transitions. Instead, the operation sequences are realized through firmware running on the processor (shown in Figure 2). Control-flow decisions are made explicitly in branches; memory references are explicit in load and store operations; sequences of computations are explicit sequences of general-purpose and application-specific computational operations.

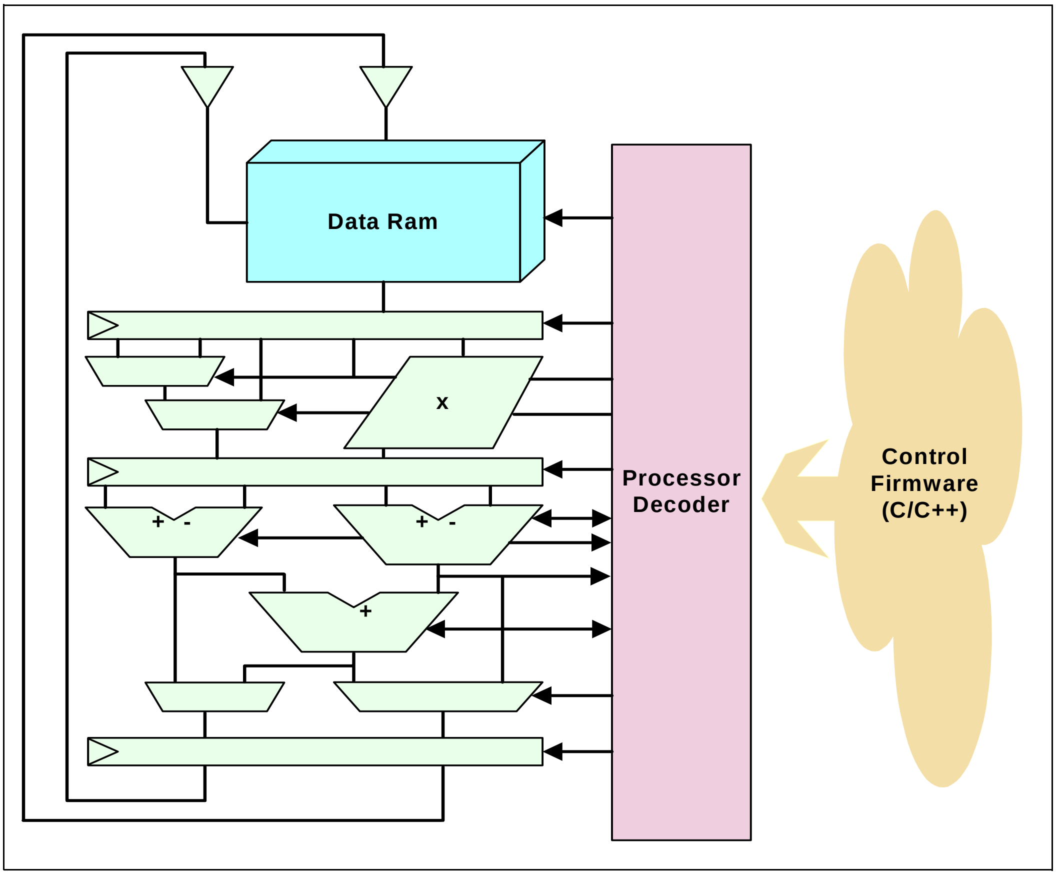


Figure 2. Programmable Function: Data Path + Processor + Firmware

The design migration from hard-wired state machine to firmware control program has important implications:

- Flexibility: Chip developers, system builders, and end-users (when appropriate) can change the block's function or add new functions just by changing the firmware.
- Software-based development: Developers can use sophisticated, low-cost software development methods to develop and debug most chip features.
- Faster, more complete system modeling: RTL simulation is slow. For a 10-million-gate design, even the fastest software-based logic simulator may not exceed a few cycles per second. By contrast, firmware simulations of extended processors running on instruction-set simulators operate at hundreds of thousands of cycles per second.

- Unification of control and data: No modern system consists solely of hard-wired logic. There is always a processor and some software or firmware somewhere in the system, if only to handle slower control and user-interface tasks. Consequently, there are always design decisions to be made about how a task should be implemented: RTL hardware or firmware. Moving most functions previously handled by RTL into processors removes the artificial distinction between control and data processing.
- Time-to-market: Moving critical functions from RTL into application-specific processors simplifies the SOC design, accelerates system modeling, and pulls in the hardware-finalization date.
- Mid-project changes: Firmware-based state machines easily accommodate changes to standards and market requirements because implementation details are not "cast in stone."
- Designer productivity: Most importantly, perhaps, migration from RTL-based design to the use of application-specific processors boosts the engineering team's productivity by reducing both the engineering manpower needed for RTL development and for verification. A processor-based SOC design approach sharply cuts risks of fatal logic bugs and permits graceful recovery when a bug is discovered.

Three examples, ranging from simple to complex, illustrate how data-path extensions allow extensible processors to replace RTL hardware in a wide variety of situations.

The first example, from the cellular telephone world, involves the GSM audio codec used in cell phones. Like many standards-based algorithms, the codec code is written in C. Profiling the codec code using an unaugmented general-purpose RISC processor revealed that 80% of the more than 200 million processor cycles needed to execute the code were devoted to executing multiplications. The simple addition of a hardware multiplier would therefore substantially accelerate this software. Tensilica's Xtensa LX processor offers a multiplier as a configuration option. That means that a designer can add a hardware multiplier to the processor's database and multiplication instructions to the processor's instruction set simply by checking a box in a configuration page of the Xtensa Processor Generator. The hardware added by checking this configuration box appears in Figure 3.

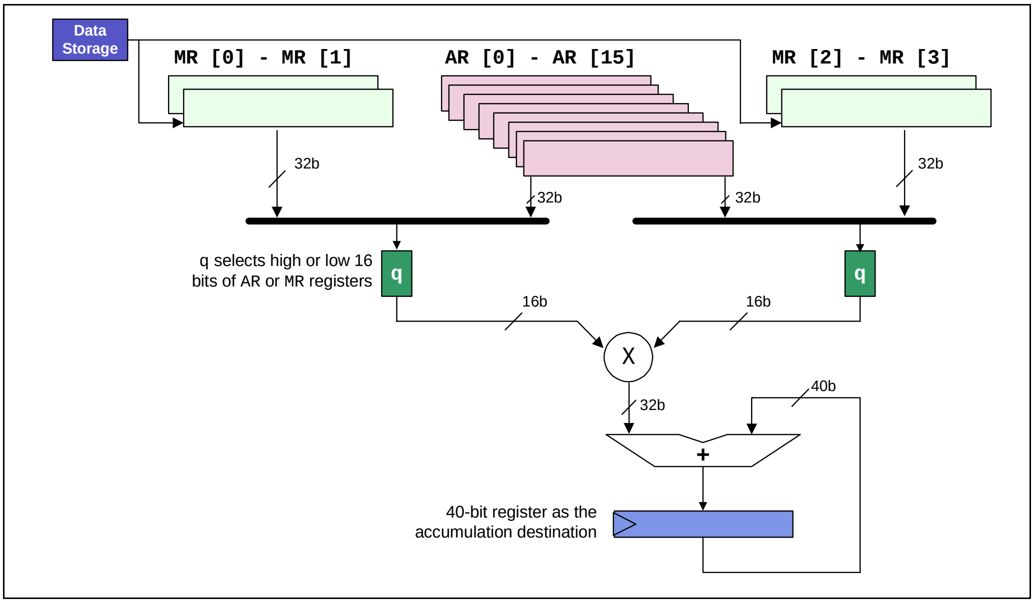

Data Storage
MR [0] - MR [1]
AR [0] - AR [15]
MR [2] - MR [3]
32b
32b
32b
32b
q selects high or low 16 bits of AR or MR registers
q
q
16b
16b
X
32b
40b
+
40-bit register as the accumulation destination

Figure 3. Multiplication Hardware Available as an Xtensa Processor Option

The addition of a hardware multiplier reduces the number of cycles needed to execute the GSM audio codec code from 204 million cycles to 28 million cycles, a 7x improvement in execution time. Adding a few more gates to the processor pipeline by selecting a multiplier/accumulator instead of a multiplier further reduces the number of cycles needed to execute the audio codec code reduced to 17.9 million cycles. Configuration options coupled with code profiling allow an SOC designer to rapidly explore a design space and to make informed cost/benefit decisions for various design approaches.

A more complex example, Viterbi decoding, also comes from GSM cellular telephony. GSM employs Viterbi decoding to pull information symbols out of a noisy transmission channel. This decoding scheme employs "butterfly" operations consisting of 8 logical operations (4 additions, 2 comparisons, and 2 selections) and uses 8 butterfly operations to decode each symbol in the received digital information stream. Typically, RISC processors need 50 to 80 instruction cycles to execute one Viterbi butterfly. A high-end VLIW DSP (TI's 320C64xx) requires only 1.75 cycles per Viterbi butterfly but it uses a lot of power to achieve this performance level. The TIE language allows a designer to add a Viterbi butterfly instruction to the Xtensa ISA. The extension added for this example uses the processor's configurable 128-bit memory bus to load the data for 8 symbols at a time. The extensions also add the pipeline hardware shown in Figure 4. The result is an average butterfly execution time of 0.16 cycles per Viterbi butterfly. An unaugmented Xtensa LX processor executes Viterbi butterflies in 42 cycles, so the butterfly hardware (approximately 11,000 gates) achieves a 250x speed improvement over the stock, unaugmented Xtensa LX processor.

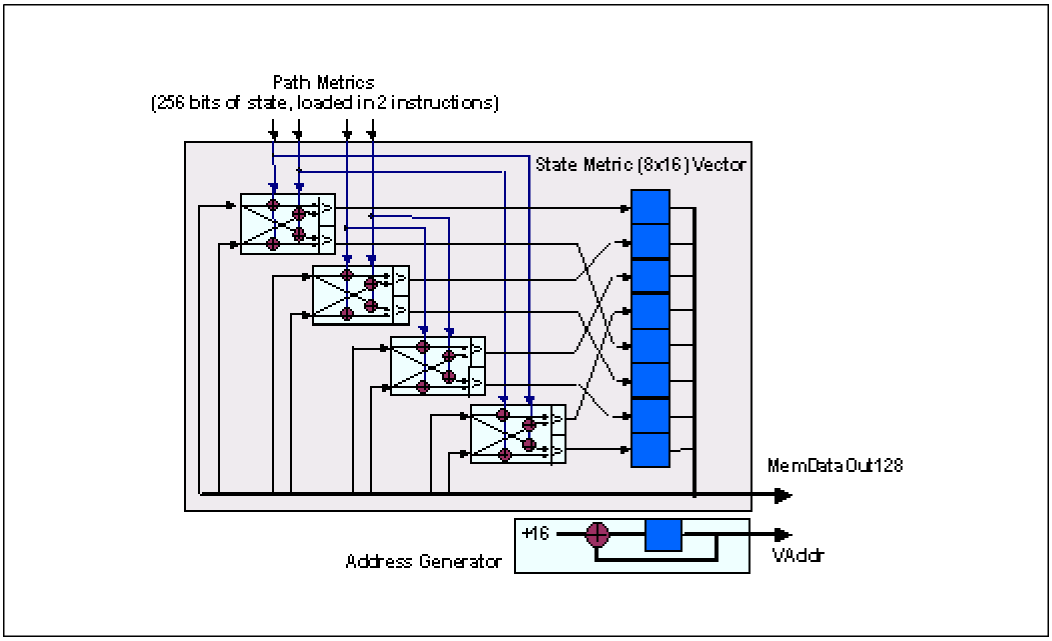

Path Metrics
(256 bits of state, loaded in 2 instructions)
State Metric (8x16) Vector
MemDataOut128
+16
Address Generator
VAddr

Figure 4. Viterbi Butterfly Pipeline Hardware

The third example, MPEG4, is from the video world. One of the most difficult parts of encoding MPEG4 video data is motion estimation, which requires the ability to search adjacent video frames for similar pixel blocks. The search algorithm employs a SAD (sum of absolute differences) operation that consists of a subtraction, an absolute value, and the addition of the resulting value with the previously computed value. For a QCIF (quarter common image format) image frame and a 15-frame/sec. image rate, the SAD operation requires slightly more than 641 million operations/sec. As shown in Figure 5, it's possible to add SIMD (single instruction, multiple data) SAD hardware capable of executing 16 pixel-wide SAD instructions per cycle to an Xtensa LX processor using TIE.

Note: Configuring the processor's bus width to 128 bits allows the processor to load 16 pixels worth of data in one instruction.

The combination of executing all three SAD component operations in one cycle and the SIMD operation that computes the values for 16 pixels in one clock cycle reduces the 641 million operations/sec. requirement to 14 million operations/sec., a 98% reduction in cycle time.

Further optimization is possible with the addition of dedicated input and output ports that completely eliminate the load/store overhead associated with formatting the data to a memory-mapped address space. The choice between using dedicated I/O ports and general-purpose memory-mapped I/O is a system-specific architectural decision.

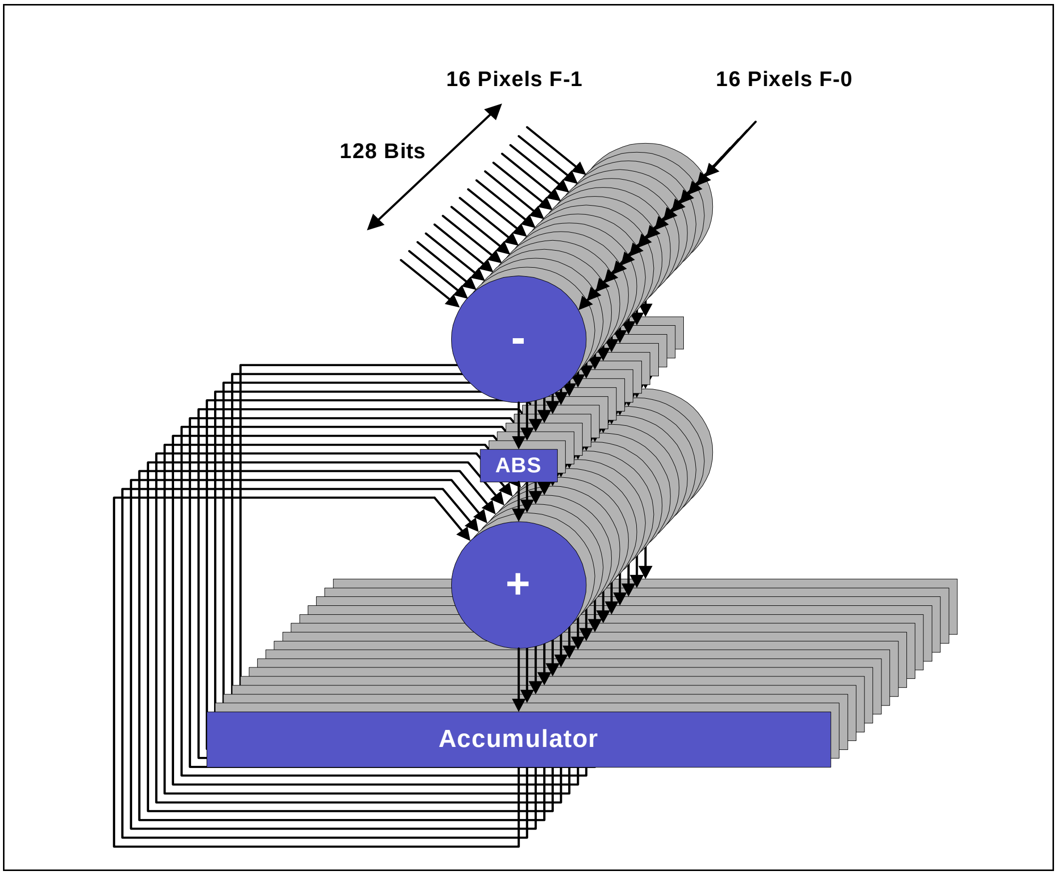


Figure 5. A SIMD SAD Computational Engine Reduces the Computational Load by 46x

Three cases where application-specific processors are not the right choice for a block's design stand out:

- Small, fixed state machines: Some logic tasks are too trivial to warrant a processor. For example, bit-serial engines such as simple UARTs fall into this category.
- Simple data buffering: Similarly, some logic tasks amount to no more than storage control. Although a processor can emulate a FIFO controller using a RAM and some wrapper logic, a hardware FIFO is faster and simpler.
- Very deep pipelines: Some computation problems have so much regularity and so little state-machine control, that one deep pipeline is the ideal implementation. The common examples—3D graphics and magnetic-disk read-channel chips—sometimes have pipelines hundreds of clock stages deep. Application-specific processors can be used to control such deep pipelines, but the benefits of instruction-by-instruction control is of less help in these applications.

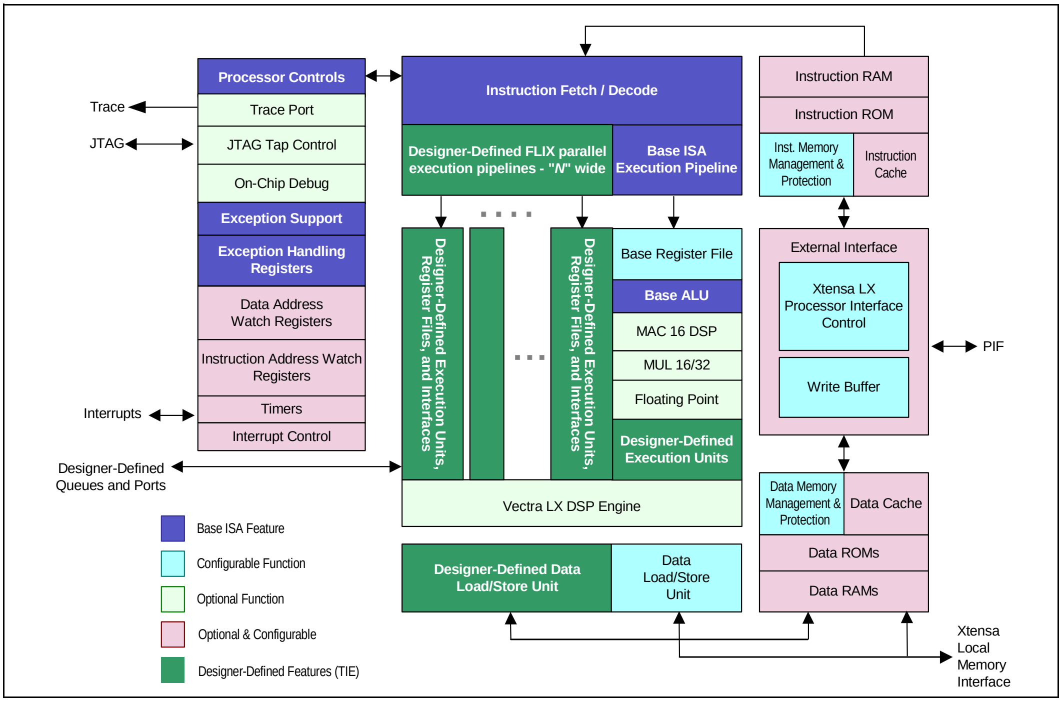
The migration of functions from hard-wired logic to software over time is a well-known phenomenon. During the 1970s and 1980s, microprocessors replaced most complex state-machine logic in a wide range of system designs. During early design exploration of pre-release protocol standards, processor-based implementations are common even for simple standards that clearly are more efficient when implemented with hard-wired logic, because software is an easier to use medium for design experimentation. Some common protocol standards that have followed this path include popular video codecs such as MPEG2, 3G wireless protocols such as W-CDMA, and cryptography and security algorithms such as SSL and triple-DES. The speed of this migration, however, has been limited by the large performance gap between software algorithms running on general-purpose processors and RTL-based designs.

The emergence of configurable and extensible application-specific processors creates a new design path that is quick and easy enough for development and refinement of new protocols and standards, fast enough to meet the application's performance demands, and efficient enough in silicon area and power to permit very high volume deployment.

1. Introduction to Xtensa LX Technology

The Xtensa LX synthesizable microprocessor core is based on Tensilica's Xtensa technology, the first configurable and extensible microprocessor architecture designed specifically to address embedded System-On-Chip (SOC) applications. The Xtensa architecture was designed from the start to be configurable, which allows designers to precisely tailor each processor implementation to match the target SOC's application requirements.

Members of the Xtensa processor family are unlike other conventional embedded processor cores—they change the rules of the SOC game. Using Xtensa technology, the system designer molds each processor to fit its assigned tasks on the SOC by selecting and configuring predefined architectural elements and by inventing completely new instructions and hardware execution units that can deliver application-specific performance levels that are orders of magnitude faster than alternative solutions. In addition to the processor core, the Xtensa Processor Generator also automatically generates a complete, optimized software-development environment that includes customized operating system support for each processor configuration. The power and flexibility of the configurable Xtensa processor family make it the ideal choice for complex SOC designs.



Processor Controls
Trace Port
JTAG Tap Control
On-Chip Debug
Exception Support
Exception Handling Registers
Data Address Watch Registers
Instruction Address Watch Registers
Timers
Interrupt Control
Trace
JTAG
Interrupts
Designer-Defined Queues and Ports
Instruction Fetch / Decode
Designer-Defined FLIX parallel execution pipelines - "N" wide
Base ISA Execution Pipeline
Designer-Defined Execution Units, Register Files, and Interfaces
Designer-Defined Execution Units, Register Files, and Interfaces
Base Register File
Base ALU
MAC 16 DSP
MUL 16/32
Floating Point
Designer-Defined Execution Units
Vectra LX DSP Engine
Designer-Defined Data Load/Store Unit
Data Load/Store Unit
Instruction RAM
Instruction ROM
Inst. Memory Management & Protection
Instruction Cache
External Interface
Xtensa LX Processor Interface Control
Write Buffer
PIF
Data Memory Management & Protection
Data Cache
Data ROMs
Data RAMs
Xtensa Local Memory Interface
Base ISA Feature
Configurable Function
Optional Function
Optional & Configurable
Designer-Defined Features (TIE)

Figure 6. The Xtensa LX Architecture: Designed for Configurability and Extensibility

As shown in Figure 6, the Xtensa LX architecture consists of various standard and configurable building blocks. Configurable function blocks are elements parameterized by the system designer. Optional function blocks indicate elements available to accelerate specific applications. The optional and configurable blocks have optional elements (such as timer interrupts and interrupt levels) that can be individually scaled to fit specific applications. SOC hardware and firmware designers can add advanced functions to the Xtensa LX architecture in the form of hardware execution units and registers to accelerate specific algorithms. Common in all configurations is the Xtensa base instruction set architecture (ISA).

Tensilica delivers five technologies to help designers build SOCs for embedded applications:

- The Xtensa LX processor architecture: a highly configurable, extensible, and synthesizable 32-bit processor. Many designer-defined, application-specific families of processors can be built around the base Xtensa ISA to optimize factors such as code size, die size, application performance, and power dissipation. Designers define new processor instructions, execution units, and registers using the Tensilica Instruction Extension (TIE) language.
- A generated software tool suite to match the configured processor architecture. This tool suite includes the Xtensa C/C++ compiler (XCC), a macro assembler, linker, debugger, diagnostics, reference test benches, and a basic software library. XCC provides C++ capabilities equivalent to the GNU C++ compiler version 2.96. It improves code performance relative to GCC in many cases and provides vectorizing DSP compiler support for the Vectra LX DSP Engine and for TIE processor extensions generated by Tensilica's XPRES compiler.
- Xtensa Xplorer, which serves as a cockpit for single- and multiple-processor SOC hardware and software design. Xtensa Xplorer integrates software development, processor optimization and multiple-processor SOC architecture tools into one common design environment. It also integrates SOC simulation and analysis tools. Xtensa Xplorer is a visual environment with a host of automation tools that makes creating Xtensa processor-based SOC hardware and software much easier. Xplorer serves as a cockpit for basic design management, invocation of Tensilica processor configuration tools (the Xtensa Processor Generator and the TIE compiler), and software development tools. Xtensa Xplorer is particularly useful for the development of TIE instructions that maximize performance for a particular application. Different Xtensa processor and TIE configurations can be saved, profiled against the target C/C++ software, and compared. Xtensa Xplorer includes automated graphing tools that create spreadsheet-style comparison charts of performance.
- A multiple processor (MP)-capable instruction set simulator (ISS) and basic simulation library.
- The XPRES compiler: a software tool capable of automatically generating performance-boosting processor extensions based on the analysis of target C or C++ application source code. The XPRES compiler sets a new benchmark for the automa-

tion of processor design and vastly improved designer productivity. This tool analyzes the target software source code, quickly generates thousands or millions of alternative processor designs, evaluates these designs for performance and gate count, selects the optimal design based on designer-defined criteria, and finally produces a description of the selected processor in TIE, which is then submitted to the Xtensa Processor Generator.

All development tools are automatically built to match the exact configuration specified in the Xtensa Processor Generator. Together, these technologies establish an improved method for rapid design, verification, and integration of application-specific hardware and software.

The TIE language is used to describe designer-defined instructions, new registers, and execution units that are then automatically added to the Xtensa processor by the Xtensa Processor Generator. As shown in Figure 7, aggressive use of parallelism and other techniques can often deliver 10X, 100X or even greater performance increases using the new TIE instructions.

Figure 7. Designer-Defined Extensions and Impact on System Performance

1.1 *Ease of Configuration with the Xtensa Processor Generator*

The Xtensa Processor Generator assists designers in creating application-specific embedded processors rapidly and reliably. By combining the Xtensa processor foundation architecture with electronic design automation (EDA) and embedded software development technology, the Xtensa Processor Generator allows developers to use the unique characteristics of their application to shape application-specific processor architecture.

Even developers with no processor design experience can use the Xtensa Processor Generator to develop highly tuned, application-specific processors for SOC designs.

The generator allows for easy selection and implementation of many processor features to match the target application. For example, a designer can define processor attributes such as exception configurations, interrupt-level settings, memory hierarchy, processor interface size, write-buffer size, on-chip RAM and ROM size, cache configurations, cache attributes, register-set size, and optional DSP functions. Additionally, designer-defined instructions can be quickly incorporated to maximize application performance and reduce code size.

1.1.1 Xtensa LX Processor Performance Summary

Tensilica's Xtensa LX processor core is specifically designed for embedded designs and it is extremely configurable, which allows designers the freedom to match the processor's capabilities to the application requirements of the target SOC. Here are some architectural details:

▪ Processor Architecture:	5- or 7-stage pipeline, high performance, 32-bit RISC
▪ Instruction Set:	Xtensa ISA with compact 16-bit and 24-bit encoding (no mode switching). Optional 32- or 64-bit FLIX instruction extensions.
▪ Performance	SOC developers using the Xtensa architecture have achieved 5X, 10X, and even 100X+ performance increases for selected algorithms by extending the processor with TIE.
▪ Clock Speed (130 nm):	350 MHz in a 130nm process, (worst-case condition, physical-synthesis flow), base configuration
▪ Size (130 nm):	0.26 mm^2 core area, base configuration 0.16 mm^2 core area, small configuration
▪ Power (130 nm):	76 μW/MHz (physical synthesis flow), base configuration 38 μW/MHz (physical synthesis flow), small configuration
▪ WAITI (stand) Power (130 nm)	4 μW/MHz (all synthesis flows), tiny configuration plus timer/interrupt configuration options 7 μW/MHz (all synthesis flows), base configuration plus timer/interrupt configuration options

Notes:

• Clock speed, core gate count, core power and total area vary with configuration, target Xtensa Processor Generator options, and choice of implementation technology.

• Core area size shown is with the Xtensa Processor Generator set for synthesis prioritization to optimize area and frequency at 200MHz.

• Use the Processor Generator estimator to estimate speed, power, and die size for your specific configuration.

• Actual power dissipation will depend on the cell libraries and process technology used to implement the processor.

1.1.2 Feature Summary

The Xtensa LX processor offers complete and robust development tools, system building blocks, and packages to develop the target embedded system-on-chip solution:

- Xtensa Processor Generator
  - Automatic and rapid generation of RTL, companion software development tools, and simulation models.
- Tensilica Instruction Extension (TIE) language
  - Designer-defined instructions are easy to construct and are automatically integrated with the base processor.
  - The Xtensa LX processor is the first member of the Xtensa processor family to feature FLIX (flexible-length instruction extensions) technology, which allows designers to create architectural additions with wide instruction words (32 or 64 bits) that have multiple instruction slots. These multiple instruction slots can contain instructions that drive the Xtensa foundation ISA as well as multiple execution units and multiple load/store units added through designer-defined TIE descriptions. FLIX instructions can be freely and modelessly intermixed in the processor's instruction stream with the Xtensa LX processor's native 24- and 16-bit instructions.
  - The Xtensa LX processor is also the first member of the Xtensa processor family to feature TIE ports and queues, which allow designers to create high-speed I/O channels that communicate directly with TIE-based execution units within the processor.
- Optional units to fit each application
  - Multipliers, 16-bit MAC, FPU, and SIMD and FLIX Vectra LX DSP Engine blocks
- Region-based memory protection
- Configurable processor attributes to fit the application
  - Big- or little-endian byte ordering
  - 5- or 7-stage pipeline to accommodate different memory speeds
  - Exceptions: non-maskable interrupt (NMI), as many as 32 external interrupts, as many as 6 interrupt priority levels, and as many as three 32-bit timer interrupts
  - 32- or 64-entry register file
  - Write Buffer: 0 (No PIF) or 4/8/16/32 entries

- Configurable Interfaces
  - 32/64/128-bit Processor Interface (PIF) width to main system memory or to an on-chip system bus. Tensilica provides a complete Vera-based tool kit for PIF bridge implementation and verification
  - “No PIF” configuration option
  - Optional high-speed Xtensa Local Memory Interface (XLMI)
  - Inbound-PIF requests allow external access to the processor’s local memory buses (the XLMI and local instruction- and data-RAM buses)
  - TIE ports and queues
- On-Chip Memory Architecture
  - Optional 1, 2, 3, and 4-way set-associative caches
  - Write-through or write-back cache-write policy
  - Cache locking per line for set-associative cache
  - Size of instruction cache:
    0/1/2/4/8/16/32 kbytes (for 1, 2, or 4-way set associative)
    0/1.5/3/6/12/24 kbytes (for 3-way set associative)
  - Instruction cache line size: 16, 32, or 64 bytes
  - Size of data cache:
    0/1/2/4/8/16/32 kbytes (for 1, 2, or 4-way set associative)
    0/1.5/3/6/12/24 kbytes (for 3-way set associative)
  - Data cache line size: 16, 32, or 64 bytes
  - Optional data RAMs and ROM and optional instruction RAMs and ROM
    Maximum memory quantities: 2 data RAMs, 2 instruction RAMs, one data ROM, and one instruction ROM
  - Size of data RAM or ROM:
    0/0.5/1/2/4/8/16/32/64/128/256 kbytes
  - Size of instruction RAM or ROM:
    0/0.5/1/2/4/8/16/32/64/128/256 kbytes
  - Optional busy signals for all local memory ports and the XLMI port

- Multiple-processor (MP) development and debug capabilities
  - MP synchronization instructions (L32AI, S32RI, and S32C1I)
  - MP capable, C/C++ callable instruction-set simulator (ISS)
  - System modeling capability via the Xtensa Modeling Protocol (XTMP) environment, which is compatible with System C
  - MP on-chip debug (OCD) capability: Trace and instruction/data breakpoint support (1 or 2 hardware-assisted instruction breakpoints and 1 or 2 hardware-assisted data breakpoints)
  - Special processor ID (PRID) register
  - GDB debugger support
  - ISS and co-simulation model (CSM) support for Mentor Graphics® Seamless™ Co-verification Environment
- Software development tools (automatically generated)
  - High-performance Xtensa C/C++ compiler (XCC) and companion GNU tool chain
- Robust EDA environment support
  - Standard and physical synthesis design flow
- Verification support
  - Comprehensive diagnostics for the Xtensa core and designer-defined TIE verification
- Clock-cycle-accurate, pipeline-modeled instruction-set simulator (ISS)
- OSKit overlays for supported real-time operating systems
  - Nucleus from Accelerated Technology®, the Embedded Systems Division of Mentor Graphics Corporation
  - WindRiver® Systems' VxWorks® for Tornado®
- Xtensa Bus Designer's Toolkit (aka PIF kit) for bus-bridge design help

1.1.3 Architectural Building Blocks

Optional and configurable blocks are elements that are optional or scalable to fit the application.

- MUL16 and MAC16
  The MUL16 option adds a 16x16-bit multiplier to the Xtensa LX processor. Source operands are drawn from the processor's main register file and the result is stored back into the same register file. The MAC16 option adds a 16x16-bit multiplier and a 40-bit accumulator, eight 16-bit operand registers (separate from the main register file), special load instructions for the operand registers, and a set of compound instructions that combine multiply, accumulate, operand load and address-update

operations. The MAC16 operand registers can be loaded with pairs of 16-bit values from memory in parallel with MAC16 operations and the MAC16 can sustain algorithms with two loads per multiply/accumulate.

- 32-bit Multiply
  The MUL32 option adds to the library of existing MAC16 (16x16-bit multiply with 40-bit accumulator) and MUL16 options. This option provides instructions that perform 32x32-bit multiplication, producing a 64-bit result.
- Multiple-processor-capable on-chip debug
  The on-chip debug (OCD) module is used to access the internal, software-visible processor state through a JTAG port. MP on-chip debug capabilities are accomplished using a serial connection to the Xtensa LX processor's TAP controllers or to custom TAP controllers. The Xtensa LX processor's OCD support includes: MP-centric break-in/break-out, debug-mode entry through exception generation, access to all program-visible registers and memory locations, execution of any instruction that the processor is configured to execute, modification of the program counter to jump to a desired code location, real-time debug, and a utility for returning to normal operating mode.
- RAM and ROM Options
  The RAM and ROM options provide internal memory ports with address ranges within the processor's address space and accessed with the same timing as cache. There are two RAM options: instruction RAM and data RAM. There are two ROM options: instruction ROM and data ROM.
- Timer Interrupt Option
  This option adds as many as three timer interrupts. The option creates one 32-bit read/write `CCOUNT` register that increments every clock cycle and as many as three 32-bit comparison registers that can generate level-1 interrupts or high-priority interrupts when they match `CCOUNT`.
- Optional Floating-point unit (FPU)
  The Xtensa LX FPU is an optional, 32-bit, single-precision, floating-point coprocessor optimized for printing, graphics, and audio applications. This IEEE Std 754-compatible coprocessor provides the programming ease and headroom of floating-point computations with fast execution speed equal to that of fixed-point processing.
- Optional Vectra LX DSP engine
  The optional Vectra LX DSP Engine is optimized to handle high-performance digital signal processing applications using fixed-point arithmetic. This SIMD (single-instruction, multiple-data) DSP option is ideal for communications, audio, and imaging applications employing a highly efficient and easy-to-program vector architecture. The Vectra LX DSP engine provides higher data throughput, lower power dissipation, and better DSP performance per watt and per area than any other processor core. The Xtensa LX processor with the Vectra LX option offers, for the first time, a single core architecture that can be rapidly configured to satisfy the specific requirements of any embedded application including control, protocol, signal, and image

processing. Like all Xtensa LX processor options, the Vectra LX DSP engine is fully-supported by software-development tools including the vectorizing Xtensa C and C++ Compiler, assemblers, simulators, RTOSes, and optimized DSP libraries.

- Xtensa Local Memory Interface
  The optional Xtensa Local Memory Interface (XLMI) port provides an extra set of address/data/control lines as a fast, alternate port. The XLMI port can be used to connect the processor to fast local memories, peripherals, and on-chip communications links. The Xtensa LX instruction-set simulator (ISS) and co-simulation model fully support the Xtensa LX XLMI port option.

1.1.4 Creating a Single Configuration Database with the Xtensa Processor Generator

The Xtensa Processor Generator uses the designer's inputs to create a processor-configuration database, as shown in Figure 8. The generator uses this database to configure the software tools, the ISS, and the processor HDL. It also uses this database to estimate the processor's area, power dissipation, and timing. These generator outputs can be used for simulation, software debugging and profiling, logic synthesis, placement and routing, timing verification, and hardware profiling.

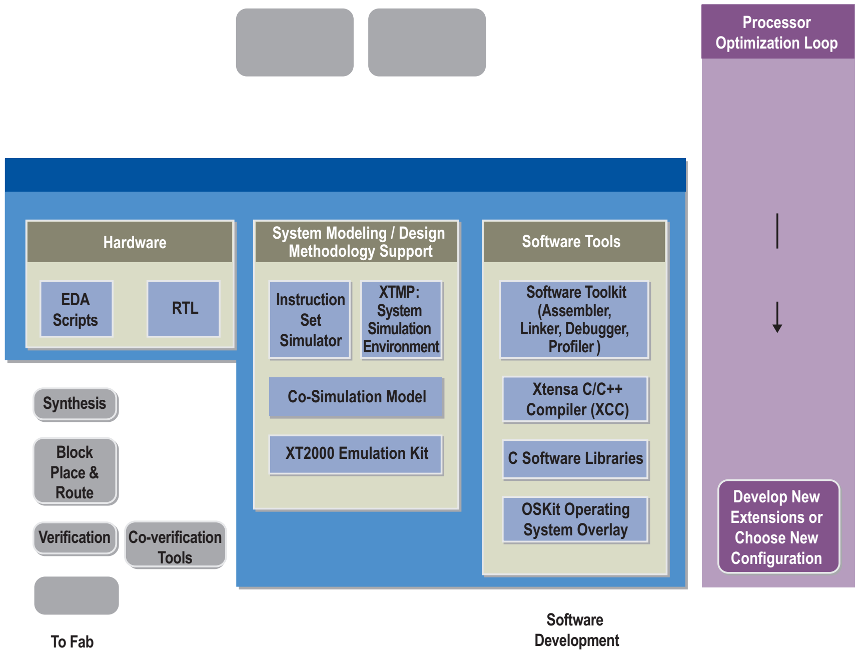

Processor Optimization Loop
Hardware
EDA Scripts
RTL
System Modeling / Design Methodology Support
Instruction Set Simulator
XTMP: System Simulation Environment
Co-Simulation Model
XT2000 Emulation Kit
Software Tools
Software Toolkit (Assembler, Linker, Debugger, Profiler)
Xtensa C/C++ Compiler (XCC)
C Software Libraries
OSKit Operating System Overlay
Develop New Extensions or Choose New Configuration
Synthesis
Block Place & Route
Verification
Co-verification Tools
To Fab
Software Development

Figure 8. The Processor Optimization Flow

1.1.5 Area, Power, and Timing Estimator

An interactive, on-line processor estimator for area, power dissipation, and timing guides the processor-definition process. Tensilica has created a comprehensive processor database based on hundreds of post-place-and-route results using a variety of commercial cell libraries and fabrication processes. The system designer selects the target ASIC technology (130 or 180 nm), and the estimator generates and displays a real-time report with estimates of the processor area in gates, the maximum clock rate in MHz, and the power dissipation in mW.

Note: The area information displayed by the estimator is an indication of the equivalent gates necessary to build the processor. This estimate does not include the size of the various memory subsystems. The designer is free to choose any available memory generator in the target technology. Tensilica does not provide memory blocks.

- The estimated gate counts reported are derived from post-synthesis design databases generated by the Synopsys Design Compiler, which reports cell area for a given netlist expressed in physical area units (square microns). Design Compiler computes the cell-area value from the total physical area for all instances of all cells in the subject netlist using the cell area listed in the selected cell library. Tensilica's estimator reports the estimated gate count for representative standard-cell technologies in each fabrication process technology. The data reported is a blended average of a number of different process technologies spanning multiple wafer foundries and cell library vendors.

  Design Compiler's cell area figure is converted to gates using the following formula:

  [reported cell area]/[area of a 2-input NAND gate with 2X drive strength]=estimated gate count

- The estimated maximum clock frequency is derived from post-layout design data. Tensilica completed more than 200 different physical implementations of the Xtensa LX processor and performed 3D extraction to calculate actual performance in creating the database. The reported data is a blended average of collected data points, extrapolated or interpolated for the specific configuration features selected.

The reported estimated power dissipation is derived from numerous simulation runs using typical and worst-case conditions and netlists synthesized at wc/wc. The reported data is a blended average of the different collected data points, extrapolated or interpolated for the specific configuration features selected.

The Xtensa Processor Generator gives real-time estimates of area, power, and timing of the target application-specific processor.

1.2 Ease of Integration

To facilitate SOC development, the Xtensa Processor Generator automatically generates the following outputs: the RTL simulation model, netlist, static-timing model, hardware-design interface, configured software tool chain, profiler, initialization and self-test code, customized instruction-set simulator, customized co-simulation model (CSM) binaries for third-party co-verification tools, third-party RTOS support, and a validation diagnostic suite. Tensilica supports the major EDA vendor tools used in industry-standard semiconductor design flows.

1.2.1 Currently Supported EDA Tools

- Simulation
  - Synopsys VCS
  - Cadence Verilog XL
  - Cadence NC-Verilog
  - Mentor Graphics MTI/VHDL
- Synthesis
  - Synopsys Design Compiler and DC Ultra
- Place & Route
  - Cadence Silicon Ensemble
  - Synopsys Apollo
  - Cadence Fire & Ice
- Unified synthesis and place & route (physical synthesis) flow
  - Synopsys Physical Compiler
  - Cadence PKS
- Static timing verification
  - Synopsys PrimeTime
- Verification
  - Synopsys VERA, DFT Compiler
- Power
  - Synopsys Power Compiler, Design Power
- Hardware/Software Co-Verification
  - Mentor Graphics Seamless
- Test
  - Synopsys DFT Compiler
  - Synopsys TetraMAX ATPG

1.3 Specification Highlights

The Xtensa Processor Generator is used to create tailored versions of the Xtensa LX processor core for target applications. When creating these tailored processors, designers must account for several factors such as target libraries, process technologies, and memory compilers. They must also keep in mind their specific performance, gate count, and power dissipation goals for the final processor cores.

The configurations shown in Table 2 are examples that illustrate the relative performance, gate counts, and power dissipation of two processor configurations. The intent of Table 2 is to show how the Xtensa LX processor attributes vary with the different configuration options. Table 3 and Table 4 highlight the performance, gate count, and power dissipation for the "base Xtensa LX configuration" and the "small Xtensa LX configuration" for standard EDA or physical synthesis flows. Although the examples shown in the following tables call the processors "base" and "small," you should not infer anything about the processors' abilities to run code based on those names.

Table 2. Example Xtensa LX Core Configurations

Processor Attributes	Base Xtensa LX Configuration	Small Xtensa LX Configuration
	Execute base ISA	Small engine to execute task code
Processor Interface	32-bit	No PIF Option
Full 16/24-bit ISA	Yes	No Loop Option No 16-bit Code
Clock Gating	Yes	Yes
Registers	32	32
Register type	Latches	Latches
Memory subsystem	1KB I-cache 1KB D-cache	4KB I-RAM, 512 B D-RAM
Optional Blocks	No MAC16 No MUL32	No MAC16 No MUL32
Vectra	No	No
TIE	No	No
Interrupts	No	No
Timers	No	No
Misc Reg	No	No
Xtensa Exception Architecture 2 (XEA2)	Region protection	Region protection
OCD	No	No
Byte Order	Little Endian	Little Endian
FPU	No	No

Table 3. Example Base Xtensa LX Configuration Specifications for Physical Synthesis Flow

130 nm, low threshold voltage (LV), worst-case process			
Parameter	**Min**	**Max**	**Unit**
Frequency of Operation		350	MHz
Power Dissipation		76	μW/MHz
Gate Count	27.5	36.5	K Gates

Table 4. Example Small Xtensa LX Configuration for Physical Synthesis Flow

130 nm, low threshold voltage (LV), worst-case process			
Parameter	**Min**	**Max**	**Unit**
Frequency of Operation		390	MHz
Power Dissipation		38	μW/MHz
Gate Count	9	25	K Gates

Notes for Tables 2 through 4:

- Frequency of operation specification is for worst case condition; the Xtensa Processor Generator is optimized for speed.
- Power dissipation is measured at typical conditions, on a netlist synthesized at worst-case condition/worst-case process.
- The performance data shown above is representative of commodity silicon processes for a wide range of Xtensa configurations. Actual performance may be faster or slower and will vary based upon choice of std-cell library, silicon process, design flow integration, physical implementation methodology and processor configuration.

1.4 Processor Performance and the Choice of Memory Subsystem

A large component of the overall SOC performance will depend on the Xtensa LX core and the memory subsystem chosen for the chip. The system designer’s choices for the memory compiler vendor, memory size, memory attributes, and fabrication technology will determine if the resultant memory subsystem specification is a match for the set-up and hold specifications for the Xtensa LX core memory interfaces.

Memory compiler vendors provide a wide variety of choices for SOC designers. Memory types can range from SRAM to register-file class. Memory product families range from high-speed to high-density offerings. There are dual-port and single-port memories. There are many specific memory attributes for a particular memory size such as the number of bits, the number of words, and the column-multiplexing factor. All of these attributes affect important memory specifications such as address set-up time (T_{ac}) and clock-to-Q output time (T_{cq}). The system designer must determine if these memory specifications meet the Xtensa LX processor’s memory-interface timing requirements.

The Xtensa LX processor provides a pipeline-depth configuration option to accommodate slower memories. The processor's 5-stage pipeline accesses memory during two stages. Each of these stages can be optionally extended by one clock, producing a 7-stage processor. These extra two stages provide more time for memory accesses. By using this option, the designer may be able to run the processor at higher clock rates while using slower memories, thus improving overall system performance.

1.5 Code Density

A 24/16-bit instruction set with a rich set of operations greatly reduces the size of Xtensa LX application code compared to conventional 32-bit RISC code. This is an especially important attribute of the Xtensa LX processor core because most of the processor cores on an SOC have on-chip memory. So small code size helps to reduce on-chip memory requirements. The Xtensa ISA optimizes the size of the program instructions by minimizing both the static number of instructions (the instructions that constitute the application program) and the average number of bits per instruction. The use of 24- and 16-bit instruction words, the use of compound instructions, the richness of the comparison and bit-testing instructions, zero-overhead-loop instructions, register windowing, and the use of encoded immediate values all contribute to the Xtensa LX processor's small code size.

The Xtensa ISA seamlessly and modelessly incorporates 16-bit instruction words that can be freely intermixed with 24-bit instructions to achieve higher code density without compromising application performance. The 16-bit instructions provide a compact encoding of the most frequently used 24-bit instructions so they are far more capable than the instructions available with today's 16-bit, fixed-ISA processors. The Xtensa LX processor's instruction stream yields an average of 20 bits per machine instruction because more than 50% of the typical static instruction mix can use the 16-bit instruction format.

The Xtensa LX processor also has several compound instructions that reduce the instruction count required to encode and execute a program. Compare-and-branch instructions (for example, branch to PC+offset if the first register is less than the second register) constitute the most important class of compound instructions. By itself, the compare-and-branch compound-instruction class reduces code size by at least a 5%. Other compound Xtensa LX instructions include shift (by 1–3 places), add/subtract, and shift-and-mask.

The Xtensa LX processor's compare instructions are richer than similar instructions found in most other processor ISAs. Included are operations such as branch if two registers are nonzero (branch if any bit set), branch if a specific bit (given by a bit number in a register or by an immediate value) is set or clear. Shift and mask further extends the bit-handling capability of the Xtensa ISA.

The Xtensa LX processor employs a feature common to DSPs but not on general-purpose architectures: zero-overhead loops—the ability to iterate a series of instructions without a branch at the end to loop back. With this feature, the Xtensa LX processor can execute loops without stalls caused by branch mis-predictions or the need for extra instructions to decrement and test the loop counter. Reducing loop overhead improves performances and reduces code size.

The Xtensa LX processor employs register windows to reduce the number of instruction bits needed to specify a register. Because most instructions specify three registers (two source register and one destination register for the result), register windowing results in substantial savings in code size. Xtensa register windows support a variable window increment size to allow call levels to completely fit into the Xtensa LX processor's 32- or 64-entry general-purpose AR register file, thus minimizing the number of stack operations required to save and restore registers around call sites. The Xtensa LX processor delays window overflow until absolutely necessary, creating fewer register-spill traps and smaller code size compared to other register-window architectures. It also means lower memory traffic and smaller code size than other non-register window architectures.

Many of the constant operands (immediate values) specified in the Xtensa LX instruction word are encoded to provide the most popular values. For example, the ADDI.N instruction (a 16-bit add with a 4-bit immediate value encoded in the instruction) provides for constants –1 and 1..15. (Note that the unneeded value of zero is replaced by the very important value of -1 in the encoding of the immediate value.) The compare-and-branch-immediate instructions similarly provide tests for the most common constants. Load and store instructions shift the constant offset appropriately for the load or store operand size.

The Xtensa ISA delivers highly efficient code that is as much as 50% smaller than today's popular RISC and CISC architectures.

1.6 FLIX (Flexible Length Instruction Xtensions)

All Xtensa processors share a common base of 16/24-bit instructions and the ability to extend the processor's ISA. The Xtensa LX processor adds FLIX (Flexible Length Instruction Xtensions) technology, which allows modeless blending of designer-defined, VLIW-style 32- or 64-bit instructions. FLIX technology allows designer-defined instructions to issue multiple operations per instruction, which increases the processor's parallel-execution abilities and further boosts application performance.

2. Xtensa Instruction Set Architecture Highlights

The Xtensa instruction set is the product of extensive research into the right balance of features to best address the needs of the embedded processor market. It borrows the best features of established RISC architectures and adds new Instruction Set Architecture (ISA) developments of its own. While the Xtensa ISA derives most of its features from RISC, it has targeted areas in which older CISC architectures have been strongest, such as compact code.

The Xtensa core ISA is implemented as a set of 24-bit instructions that perform 32-bit operations. The instruction width was chosen primarily with code-size economy in mind. The instructions themselves were selected for their utility in a wide range of embedded applications. The core ISA has many powerful features, such as compound operation instructions that enhance its fit to embedded applications, but it avoids features that would benefit some applications at the expense of cost or power on others (for example, features that require extra register-file ports). Such features can be implemented in the Xtensa architecture using options and coprocessors specifically targeted at a particular application area.

The Xtensa ISA is organized as a core set of instructions with various optional packages that extend the functionality for specific application areas. This allows the designer to include only the required functionality in the processor core, maximizing the efficiency of the solution. The core ISA provides the functionality required for general control applications, and excels at decision-making and bit and byte manipulation. The core also provides a target for third-party software, and for this reason deletions from the core are not supported. On the other hand, numeric computing applications such as Digital Signal Processing are best done with optional ISA packages appropriate for specific application areas, such as the MAC16 option for integer filters, or the Floating-Point Coprocessor option for high-end audio processing.

2.1 Registers

The Xtensa LX processor contains a variety of registers and register files:

- 32-bit general-purpose register file with register windows
- 32-bit program counter (PC)
- Various special registers (see Table 5) including a shift-amount register (SAR)
- Additional designer-defined registers
- 16 optional 1-bit Boolean registers
- 16 optional 32-bit floating-point registers

- 4 optional 32-bit MAC16 data registers
- Optional Vectra LX DSP register files

The Xtensa ISA defines a general-purpose AR register file consisting of 32 or 64 registers. Instructions have access to this register file through a sliding “window” of 16 registers. (Register operand encoding for source and destination registers is 4 bits.) A function call can rotate the window by 4, 8, or 12 registers, where the amount of rotation is determined by the call opcode. Figure 9 illustrates a window that has been rotated by 8 registers. There is always some overlap with the previous window, so that function arguments and return values can be passed in the overlapping registers.

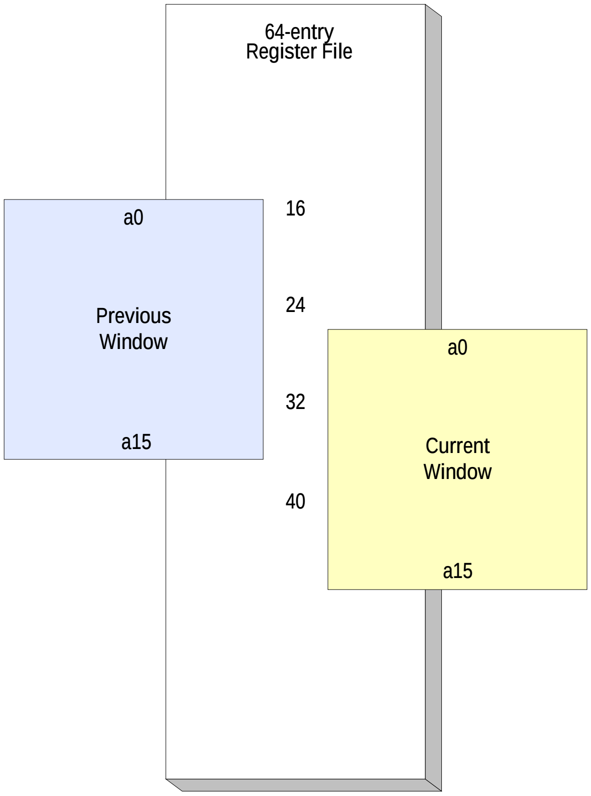


Figure 9. Register Windows

After rotating to a new window, the code has access to a new set of registers that can be freely used without having to save and restore the previous contents at function-call boundaries. Thus, the primary benefit of register windows is to reduce code size by eliminating the need for instructions to save and restore register values while limiting the number of bits used to encode the register addresses, which reduces the overall instruction size, saving code space and improving execution performance.

Eventually, the register window overflows when it rotates to a position in the AR register file that already holds the contents of a previous window. This overflow generates a window-overflow exception, and the exception handler spills the contents of the older window to a preallocated spill area on the stack. When returning from a function call and rotating back to a window that has been spilled, a window-underflow exception occurs. The underflow exception handler copies the register values from the stack back into the register file.

Only the general-purpose AR register file employs register windows. Additional register files added through Xtensa LX configuration options and register files created through Tensilica's TIE extension language do not employ register windowing.

In addition to the general-purpose AR register file, the Xtensa LX processor has a number of other architectural and special-purpose registers, listed in Table 5. Special registers are accessed with dedicated "read special register" (`RSR`), "write special register" (`WSR`), and "exchange special register" (`XSR`) instructions. Coprocessor registers and user registers (created with the TIE processor-extension language) are accessed with dedicated "read user register" (`RUR`), "write user register" (`WUR`) instructions. Note that special registers and coprocessor/user registers are numbered and these numbers may overlap in some cases, although this overlap does not cause a problem because these two sets of registers are accessed with different instructions.

Table 5. Alphabetical List of Xtensa LX Registers

Name[1]	Description	Package (Core or Option)	Special Register Number
`ACCHI`	Accumulator high bits	MAC16 option	17
`ACCLO`	Accumulator low	MAC16 option	16
`AR`	General-purpose register file	Core architecture	n/a
`BR`	Boolean registers	Boolean option	4
`CCOMPARE0..3`	Cycle number to generate interrupt	Timer Interrupt option	240-243
`CCOUNT`	Cycle count	Timer Interrupt option	234
`CPENABLE`	Coprocessor enable bits	Coprocessor option	224
`DBREAKA0..15`	Data break address	Debug option	144-159
`DBREAKC0..15`	Data break control	Debug option	160-175
`DEBUGCAUSE`	Cause of last debug exception	Debug option	233
`DDR`	Debug data register	Debug option	104
`DEPC`	Double exception PC	Exception option	192
`EPC1`	Level-1 exception PC	Exception option	177
`EPC2..15`	High-level exception PC	High Priority Interrupt option	178-191
`EPS2..15`	High-level exception PS	High Priority Interrupt option	194-207

Table 5. Alphabetical List of Xtensa LX Registers (continued)

Name[1]	Description	Package (Core or Option)	Special Register Number
`EXCCAUSE`	Cause of last exception/level-1 interrupt	Exception option	232
`EXCSAVE1`	Level-1 exception save location	Exception option	209
`EXCSAVE2..15`	High-level exception save location	High Priority Interrupt option	210-223
`EXCVADDR`	Exception virtual address	Exception option	238
`FCR`	Floating-point control register	Floating-Point Coprocessor option	User[2] 232
`FR`	Floating-point registers	Floating-Point Coprocessor option	n/a
`FSR`	Floating-point status register	Floating-Point Coprocessor option	User[2] 233
`IBREAKA0..15`	Instruction break address	Debug option	128-143
`IBREAKENABLE`	Instruction break enable bits	Debug option	96
`ICOUNT`	Instruction count	Debug option	236
`ICOUNTLEVEL`	Instruction count level	Debug option	237
`INTENABLE`	Interrupt enable bits	Interrupt option	228
`INTCLEAR`	Clear request bits in `INTERRUPT`	Interrupt option	227
`INTERRUPT`	Read interrupt request bits[3]	Interrupt option	226
`INTSET`	Set request bits in `INTERRUPT`[3]	Interrupt option	226
`LBEG`	Loop-begin address	Loop option	0
`LCOUNT`	Loop count	Loop option	2
`LEND`	Loop-end address	Loop option	1
`LITBASE`	Literal base	Extended L32R option	5
`MISC0..3`	Misc register 0-3	Miscellaneous Special Registers option	244-247
`M0..3`	MAC16 data registers	MAC16 option	32-35
`PC`	Program counter	Core Architecture	n/a
`PRID`	Processor ID	Processor ID option	235
`PS`	Processor state	Various	230
`SAR`	Shift-amount register	Core Architecture	3
`SAV`	Shift-amount valid	Speculation option	11
`SCOMPARE1`	Expected data value for S32C1I	Multiprocessor Synchronization option	12
`WindowBase`	Base of current AR window	Windowed Register option	72
`WindowStart`	Call-window start bits	Windowed Register option	73

1 Used in `RSR`, `WSR`, and `XSR` instructions.

2 `FCR` and `FSR` are User Registers where most are System Registers. These names are used in `RUR` and `WUR` instructions.

3 The INTERRUPT and INTSET registers share special register number 226. An RSR instruction directed at register 226 reads the INTERRUPT register, and a WSR instruction directed at register 226 writes to the INTSET register. Do not use the XSR instruction with special register 226.

2.2 Memory Addressing

The Xtensa ISA employs 32-bit memory addressing (a 2^{32} or 4Gbyte address space). Load and store instructions can access the entire 32-bit memory space. The Xtensa LX program counter (PC) also holds a 32-bit address and can address instructions throughout the entire 4Gbyte memory space. However, the register-window call and return instructions cannot cross 1Gbyte address boundaries because register-window call instructions only store the lower 30 bits of the return address and register-window return instructions leave the high two bits of the PC unchanged.

Xtensa LX processors can be configured for either little-endian (PC-style) or big-endian (internet-style) byte order. Figure 10 illustrates these two different byte orderings. With big-endian byte ordering, the most significant byte in a 32-bit value resides at the lowest address. With little-endian ordering, the least significant byte in a 32-bit value resides at the lowest address.

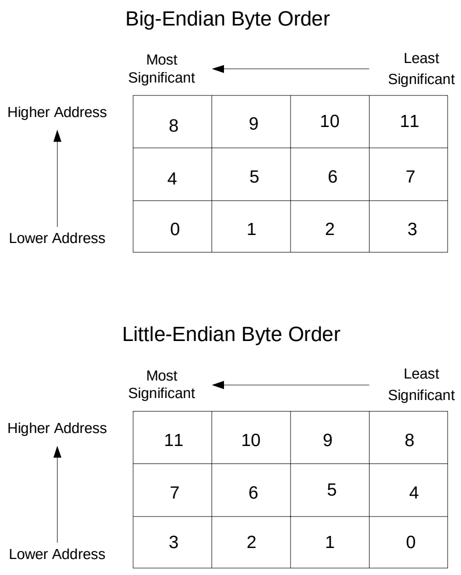


Figure 10. Big- and Little-Endian Byte Ordering

2.3 Memory References

The Xtensa LX processor accesses memory as bytes (8 bits), half-words (16 bits) and words (32 bits). In addition, designer-defined 64- and 128-bit memory references can be added using the Tensilica Instruction Extension (TIE) language.

Table 6. Core Instruction Summary (continued)

Instruction Category	Instructions

Shift	`EXTUI, SRLI, SRAI, SLLI`
	`SRC, SLL, SRL, SRA`
	`SSL, SSR, SSAI, SSA8B, SSA8L`

Processor Control	`RSR, WSR, XSR, RUR, WUR, ISYNC, RSYNC, ESYNC, DSYNC`

Table 7. Core Architecture Instructions

Instruction	Definition
ABS	Absolute value
ADD	Add two registers
ADDI	Add a register and an 8-bit immediate
ADDMI	Add a register and a shifted 8-bit immediate
ADDX2/4/8	Add two registers with one of them shifted left by one/two/three
AND	Bitwise AND of two registers
BALL/BANY	Branch if all/any bits specified by a mask in one register are set in another register
BBC/BBS	Branch if the bit specified by another register is clear/set
BBCI/BBSI	Branch if the bit specified by an immediate is clear/set
BEQ/BEQI	Branch if a register equals another register/encoded constant
BEQZ	Branch if a register equals Zero
BGE/BGEI	Branch if one register is Greater than or Equal to a register/encoded constant
BGEU/BGEUI	Branch if one register is Greater than or Equal to a register/encoded constant as unsigned
BGEZ	Branch if a register is Greater than or Equal to Zero
BLT/BLTI	Branch if one register is Less than a register/encoded constant
BLTU/BLTUI	Branch if one register is Less than a register/encoded constant as unsigned
BLTZ	Branch if a register is Less than Zero
BNALL/BNONE	Branch if some/all bits specified by a mask in a register are clear in another register
BNE/BNEI	Branch if a register does not equal a register/encoded constant
BNEZ	Branch if a register does not equal Zero
CALL0	Call subroutine at PC plus offset, place return address in A0
CALLX0	Call subroutine register specified location, place return address in A0
DSYNC/ESYNC	Wait for data memory/execution related changes to resolve
EXTUI	Extract field specified by immediates from a register
ISYNC	Wait for Instruction-Fetch-related changes to resolve
J	Jump to PC plus offset

Table 7. Core Architecture Instructions (continued)

Instruction	Definition
JX	Jump to register-specified location
L8UI	Load zero-extended byte
L16SI/L16UI	Load sign/zero-extended 16-bit quantity
L32I	Load 32-bit quantity
L32R	Load literal at offset from PC
MEMW	Wait for any possible memory ordering requirement
EXTW	Wait for any possible external ordering requirement
MOVEQZ	Move register if the contents of a register are Equal to Zero
MOVGEZ	Move register if the contents of a register are Greater than or Equal to Zero
MOVI	Move a 12-bit immediate to a register
MOVLTZ	Move register if the contents of a register are Less than Zero
MOVNEZ	Move register if the contents of a register are not Zero
NEG	Negate a register
OR	Bitwise OR two registers
RET	Subroutine Return through A0
RSR.*	Read a Special Register
RSYNC	Wait for dispatch related changes to resolve
S8I/S16I/S32I	Store byte/16-bit quantity/32-bit quantity
SLL/SLLI	Shift Left/Right Logical by SAR/Immediate
SRA/SRAI	Shift Right Arithmetic by SAR/Immediate
SRC	Shift Right Combined by SAR with two registers as input and one as output
SRL/SRLI	Shift Right Logical by SAR/Immediate
SSA8B/SSA8L	Use low 2-bits of address register to prepare SAR for SRC assuming big/little endian
SSAI	Set SAR to Immediate Value
SSL/SSL	Set SAR from register for Left/Right Shift
SUB	Subtract two registers
SUBX2/4/8	Subtract two registers with the un-negated one shifted left by one/two/three
WSR	Write a Special Register
XOR	Bitwise XOR two registers
XSR	Read and write a Special Register in an exchange

2.5.1 Load/Store Instructions

Xtensa LX load instructions (listed in Table 8) form an address by adding a base register and an 8-bit unsigned offset. The address is then used to access the memory system (sometimes through a cache). The memory system returns a data item (either 32, 64, or 128 bits, depending on the configuration). The load instructions then extract the referenced data from that memory item and either zero-extend or sign-extend the result to be written into a register. Unless the Unaligned Exception option is enabled, the processor does not handle misaligned data or trap when a misaligned address is used; instead, it simply loads the aligned data item at the computed address. This feature allows the funnel shifter to be used with a pair of loads to reference data on any byte address.

Only the load instructions `L32I`, `L32I.N`, and `L32R` may access instruction RAM and instruction ROM locations.

Table 8. Load Instructions

Instruction	Definition
L8UI	8-bit unsigned load (8-bit offset)
L16SI	16-bit signed load (8-bit shifted offset)
L16UI	16-bit unsigned load (8-bit shifted offset)
L32I	32-bit load (8-bit shifted offset)
L32R	32-bit load PC-relative (16-bit negative word offset)

Store instructions (listed in Table 9) are similar to load instructions in address formation. Store memory errors are not synchronous exceptions; it is expected that the memory system will use an interrupt to indicate an error on a store.

Only the store instructions `S32I` and `S32I.N` may access instruction RAM and instruction ROM.

Table 9. Store Instructions

Instruction	Definition
S8I	8-bit store (8-bit offset)
S16I	16-bit store (8-bit shifted offset)
S32I	32-bit store (8-bit shifted offset)

2.5.2 Jump and Call Instructions

The unconditional branch instruction, `J`, has a longer range (PC-relative) than conditional branches. Calls have slightly longer range because they target 32-bit aligned addresses. In addition, jump and call indirect instructions (listed in Table 10) provide support for case dispatch, function variables, and dynamic linking.

Table 10. Jump and Call Instructions

Instruction	Definition
CALL0	Call subroutine, PC-relative
CALLX0	Call subroutine, address in register
J	Unconditional jump, PC-relative
JX	Unconditional jump, address in register
RET	Subroutine return—jump to return address. Used to return from a routine called by `CALL0/CALLX0`.

2.5.3 Conditional Branch Instructions

Conditional branch instructions (listed in Table 11) compare a register operand against zero, an immediate, or a second register value and conditionally branch based on the result of the comparison. Compound compare and branch instructions improve code density and performance compared to other ISAs. All branches are `PC`-relative; the immediate field contains the difference between the target `PC` plus 4 and the current `PC`. The use of a `PC`-relative offset of -3 to 0 is illegal and reserved for future use.

Table 11. Conditional Branch Instructions

Instruction	Definition
BEQZ	Branch if Equal to Zero
BNEZ	Branch if Not Equal to Zero
BGEZ	Branch if Greater than or Equal to Zero
BLTZ	Branch if Less than Zero
BEQI	Branch if Equal Immediate
BNEI	Branch if Not Equal Immediate
BGEI	Branch if Greater than or Equal Immediate
BLTI	Branch if Less than Immediate
BGEUI	Branch if Greater than or Equal Unsigned Immediate
BLTUI	Branch if Less than Unsigned Immediate
BBCI	Branch if Bit Clear Immediate
BBSI	Branch if Bit Set Immediate
BEQ	Branch if Equal
BNE	Branch if Not Equal
BGE	Branch if Greater than or Equal
BLT	Branch if Less than
BGEU	Branch if Greater than or Equal Unsigned

Table 11. Conditional Branch Instructions (continued)

Instruction	Definition
BLTU	Branch if Less than Unsigned
BANY	Branch if Any of Masked Bits Set
BNONE	Branch if None of Masked Bits Set (All Clear)
BALL	Branch if All of Masked Bits Set
BNALL	Branch if Not All of Masked Bits Set
BBC	Branch if Bit Clear
BBS	Branch if Bit Set

2.5.4 Move Instructions

`MOVI` sets a register to a constant encoded in the instruction. The conditional move instructions (listed in Table 12) are used for branch avoidance.

Table 12. Move Instructions

Instruction	Definition
MOVI	Load register with 12-bit signed constant
MOVEQZ	Conditional move if Zero
MOVNEZ	Conditional move if non-Zero
MOVLTZ	Conditional move if Less than Zero
MOVGEZ	Conditional move if Greater than or Equal to Zero

2.5.5 Arithmetic Instructions

The arithmetic instructions (listed in Table 13) include add and subtract with a small shift for address calculations and for synthesizing constant multiplies. The `ADDMI` instruction is used to extend the range of load and store instructions.

Table 13. Arithmetic Instructions

Instruction	Definition
ADD	Add two registers $\mathtt{AR[r]} \leftarrow \mathtt{AR[s]} + \mathtt{AR[t]}$
ADDX2	Add register to register shifted by 1 $\mathtt{AR[r]} \leftarrow (\mathtt{AR[s]}_{30..0} \Vert 0) + \mathtt{AR[t]}$
ADDX4	Add register to register shifted by 2 $\mathtt{AR[r]} \leftarrow (\mathtt{AR[s]}_{29..0} \Vert 0^2) + \mathtt{AR[t]}$
ADDX8	Add register to register shifted by 3 $\mathtt{AR[r]} \leftarrow (\mathtt{AR[s]}_{28..0} \Vert 0^3) + \mathtt{AR[t]}$

SUB	Subtract two registers $AR[r] \leftarrow AR[s] - AR[t]$
SUBX2	Subtract register from register shifted by 1 $AR[r] \leftarrow (AR[s]_{30..0} \| 0) - AR[t]$
SUBX4	Subtract register from register shifted by 2 $AR[r] \leftarrow (AR[s]_{29..0} \| 0^2) - AR[t]$
SUBX8	Subtract register from register shifted by 3 $AR[r] \leftarrow (AR[s]_{28..0} \| 0^3) - AR[t]$
NEG	Negate $AR[r] \leftarrow 0 - AR[t]$
ABS	Absolute Value $AR[r] \leftarrow$ if $AR[s]_{31}$ then $0 - AR[s]$ else $AR[s]$
ADDI	Add signed constant to register $AR[t] \leftarrow AR[s] + ({imm8_7}^{24} \| imm8)$
ADDMI	Add signed constant shifted by 8 to register $AR[t] \leftarrow AR[s] + ({imm8_7}^{16} \| imm8 \| 0$

2.5.6 Bitwise Logical Instructions

The bitwise logical instructions (listed in Table 14) provide a core set of instructions from which other logical instructions can be synthesized. There are no immediate forms of these instructions, because the immediate value would be limited to four bits as a result of the Xtensa LX core’s compact 24/16-bit instruction word.

Note that the Extract Unsigned Immediate instruction (`EXTUI`) can perform many of the most commonly used bitwise-AND operations.

2.5.7 Shift Instructions

The Xtensa ISA provides conventional immediate shifts (logical left, logical right, and arithmetic right), but it does not provide single-instruction shifts in which the shift amount is a register operand. Taking the shift amount from a general register can create critical timing paths that might limit the processor's maximum clock frequency. Also, simple shifts do not extend efficiently to larger widths. Funnel shifts (where two data values are catenated on input to the shifter) solve this problem, but require too many operands. The Xtensa LX architecture solves both of these problems by providing a funnel shift in which the shift amount is taken from the SAR (shift amount) register. Variable shifts are synthesized by the Xtensa XCC compiler using an instruction to compute SAR from the shift amount in a general register, followed by a funnel shift.

To facilitate unsigned bit-field extraction, the `EXTUI` instructions take a 4-bit mask field that specifies the number of mask bits to use on the shift result. The 4-bit field specifies masks of 1 to 16 ones. The `SRLI` instruction provides shifting without a mask.

The legal range of values for the 6-bit SAR is 0 to 32, not 0 to 31. The use of SRC, SRA, `SLL`, or SRL when SAR > 32 is undefined.

The contents of the SAR are undefined after processor reset and should be initialized by the reset exception-vector code.

The Xtensa LX shift instructions (listed in Table 15) provide a rich set of operations while avoiding critical timing paths to keep clock frequencies high.

Table 15. Shift Instructions

Instruction	Definition
EXTUI	Extract Unsigned field Immediate Shifts right by `0..31` and ANDs with a mask of `1..16` ones The operation of this instruction when the number of mask bits exceeds the number of significant bits remaining after the shift is undefined and reserved for future use.
SLLI	Shift Left Logical Immediate by `1..31` bit positions
SRLI	Shift Right Logical Immediate by `0..15` bit positions There is no `SRLI` for shifts ≥ 16; use `EXTUI` instead.
SRAI	Shift Right Arithmetic Immediate by `0..31` bit positions
SRC	Shift Right Combined (a funnel shift with shift amount from SAR) The two source registers are catenated, shifted, and the least significant 32 bits returned.
SRA	Shift Right Arithmetic (shift amount from SAR)
SLL	Shift Left Logical (Funnel shift `AR[s]` and `0` by shift amount from SAR)
SRL	Shift Right Logical (Funnel shift `0` and `AR[s]` by shift amount from SAR)

Table 15. Shift Instructions (continued)

Instruction	Definition
SSA8B	Set Shift Amount Register (SAR) for big-endian byte align The `t` field must be zero.
SSA8L	Set Shift Amount Register (SAR) for little-endian byte align
SSR	Set Shift Amount Register (SAR) for Shift Right Logical This instruction differs from `WSR` to `SAR` in that only the four least significant bits of the register are used.
SSL	Set Shift Amount Register (SAR) for Shift Left Logical
SSAI	Set Shift Amount Register (SAR) Immediate

2.5.8 Reading and Writing the Special Registers

The SAR register is part of the Non-Privileged Special Register set in the Xtensa ISA (the other registers in this set are associated with the architectural options). The contents of the special register in the core architecture can be read and written to and from the AR registers with the Read Special Register (`RSR.SAR`), Write Special Register (`WSR.SAR`), and Exchange Special Register (`XSR.SAR`) instructions, as shown in Table 16.

Table 16. Reading and Writing Special Registers

Register Name	Special Register Number	RSR.* Instruction	WSR .* Instruction
SAR	3	$\texttt{AR[t]} \leftarrow 0^{26}\Vert\texttt{SAR}$	$\texttt{SAR} \leftarrow \texttt{AR[t]}_{5..0}$

The `RSR.*`, `WSR.*`, and `XSR.*` instructions are used to read, write, and exchange special registers for both the core architecture and the architectural options. They are used for saving and restoring context, processing interrupts and exceptions, and controlling address translation and attributes. The `XSR.*` instruction reads and writes both the special register, and `AR[t]`. It combines the `RSR.*` and `WSR.*` operations to exchange the special register with `AR[t]`.

2.6 A Word About Optional Architectural Extensions

Section 2.7 through Section 2.16 describe optional extensions to the base Xtensa LX processor architecture. Thinking about and designing configurable, extensible processors is not quite like thinking about or designing processors with fixed ISAs. Configurable processors cover a wider range of applications by making some features optional so that only applications that require the optional features need to implement these features. This attribute of configurable processors saves gates and reduces power.

2.7 Zero-Overhead Loop Option

The Zero-Overhead Loop option adds the ability for the Xtensa LX processor to execute a zero-overhead loop where the number of iterations (not counting an early exit) can be determined prior to entering the loop. The capability is often used for while loops and for DSP applications where branch overhead in a heavily used loop is unacceptable. A single loop instruction defines both the beginning and end of a loop as well as a loop count that sets the number of times the loop executes.

2.7.1 LCOUNT/LBEG/LEND Registers

The `LCOUNT/LBEG/LEND` registers and the `LOOP`, `LOOPNEZ`, and `LOOPGTZ` instructions are used to control zero-overhead loops.

Table 17 and Table 18 show the loop option's architectural additions.

Table 17. Loop Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
LBEG	1	32	Loop Begin	R/W	0
LEND	1	32	Loop End	R/W	1
LCOUNT	1	32	Loop Count	R/W	2

1. Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions.

`LBEG` and `LEND` are undefined after processor reset. `LCOUNT` is initialized to zero after processor reset to prevent spurious loopbacks in the reset exception vector code.

Table 18. Loop Option Instruction Additions

Instruction	Definition
LOOP	Set up a zero-overhead loop by setting `LBEG`, `LEND`, and `LCOUNT` special registers.
LOOPGTZ	Set up a zero-overhead loop by setting `LBEG`, `LEND`, and `LCOUNT` special registers. Skip loop if `LCOUNT` is not positive.
LOOPNEZ	Set up a zero-overhead loop by setting `LBEG`, `LEND`, and `LCOUNT` special registers. Skip loop if `LCOUNT` is zero.

2.7.2 Loop Instruction Alignment

There are restrictions on the alignment of the first instruction in a loop body (the instruction pointed to by the `LBEG` register) with respect to a 32-bit word (or a 64-bit double word for Xtensa LX configurations with 64-bit instruction extensions). For configurations with no wide (32- or 64-bit) instructions defined, a 16-bit instruction as the first instruction of a loop body must start at a byte offset of 0, 1, or 2 within a 32-bit word.

(**Note:** Figure 11 shows byte offsets within 32- and 64-bit words.) If the first loop-body instruction is a 24-bit instruction, it must start at byte offsets 0 or 1 within a 32-bit word. If 32-bit instructions are defined, the loop instruction-alignment rules for 16- and 24-bit instructions are unchanged and a 32-bit instruction used as the first instruction of a loop body must start at byte offset 0 within the 32-bit word.

If 64-bit instructions are defined, the instruction-alignment rules are more relaxed for the first instruction of a loop body. If 64-bit instructions are defined, 16-bit instructions can be aligned to byte offsets 0 through 6 and 24-bit instructions can be aligned at byte offsets 0 through 5 within a 64-bit double word. If a 64-bit instruction is used as the first instruction in a loop body, it must begin at byte offset 0 of a 64-bit word.

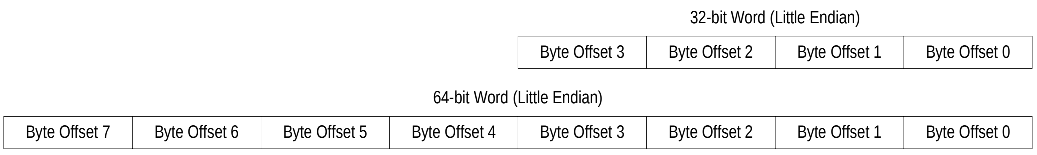


Figure 11. Byte Offsets for Zero-Overhead-Loop Instruction Alignments

The last instruction of the loop must not be a call, `ISYNC`, `WAITI`, or `RSR.LCOUNT`. A taken branch should not be used to iterate within a zero-overhead loop, because the value of `LCOUNT` is undefined if the branch is the last instruction in the loop. However, a branch instruction may be used to exit the zero-overhead loop (because the value of `LCOUNT` is irrelevant in this situation).

2.7.3 Loops Disabled During Exceptions

Loops are disabled when `PS.EXCM` is set (indicating that an exception has occurred). This mechanism prevents program code from maliciously or accidentally setting `LEND` to an address in an exception handler and then causing the exception, which would then cause the exception handler to misexecute.

2.7.4 Loopback Semantics

The LX processor's design includes the following to compute the PC of the next instruction:

```
if LCOUNT ≠ 0 and CLOOPENABLE and nextPC = LEND then
    LCOUNT ← LCOUNT − 1
    nextPC ← LBEG
endif
```

The semantics above have some non-obvious consequences. A taken branch to the address in LEND does not cause a transfer to LBEG. Thus, a taken branch to the LEND instruction can be used to exit the loop prematurely. To conditionally begin the next loop iteration, a branch to a NOP before LEND may be used.

2.8 Extended L32R Option

The Extended L32R option increases the reach of the standard L32R instruction by adding a base register for access to literal data. The standard, PC-relative L32R instruction has an offset that reaches as far as 256 kbytes below the current PC. When an instruction RAM approaches or exceeds 256 kbytes, accessing literals becomes more difficult. The Extended L32R option eases access to distant literal data by replacing the PC-relative operation of the L32R instruction with a similar function based on the address in the literal-base register.

2.8.1 Extended L32R Architectural Additions

Table 19 shows the Extended L32R option's architectural additions.

Table 19. Extended L32R Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
LITBASE	1	21	Literal Base	R/W	5

1. Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions.

2.8.2 The Extended L32R Literal Base Register

The Literal Base (LITBASE) register (shown in Figure 12) contains one enable bit (En) and 20 upper bits that define the location of the literal base. When the enable bit is clear, the L32R instruction loads a literal at a negative offset from the PC. When the enable bit is set, the L32R instruction loads a literal at a positive offset from the address formed by the 20 upper bits of Literal Base and 12 lower bits of 12'h000.

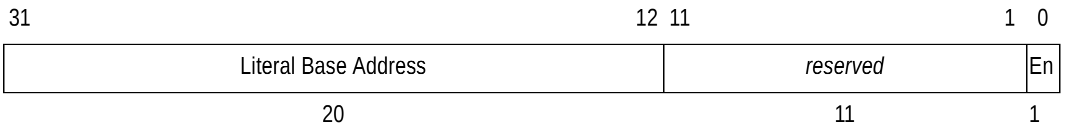


Figure 12. LITBASE Register Format

2.9 16-bit Integer Multiply Option

The MUL16 option provides two instructions that perform 16×16 multiplication, producing a 32-bit result. It is typically useful for Digital Signal Processing (DSP) algorithms that require 16 bits or less of input precision (32 bits of input precision is provided by the 32-bit Integer Multiply option) and do not require more than 32-bit accumulation (as provided by the MAC16 option). A 16×16 multiplier consumes one fourth the area of a 32×32 multiplier, so the MUL16 option is less costly (in terms of gates or silicon) than the 32-bit Integer Multiply MUL32 option. Because it lacks an accumulator and data registers, the MUL16 option is also less costly than the MAC16 option.

2.9.1 16-bit Integer Multiply Architectural Additions

Table 20 shows the MUL16 option's architectural additions. There are no configuration parameters associated with the MUL16 option and no processor state is added.

Table 20. 16-bit Integer Multiply Option Instruction Additions

Instruction	Definition
MUL16S	Signed 16×16 multiplication of the least-significant 16 bits of AR[s] and AR[t], with the 32-bit product written to AR[r]
MUL16U	Unsigned 16×16 multiplication of the least-significant 16 bits of AR[s] and AR[t], with the 32-bit product written to AR[r]

2.10 MAC16 Option

The MAC16 option adds multiply-accumulate functions that are useful in DSP and other media-processing operations. This option adds a 40-bit Accumulator (`ACC`), four 32-bit Data Registers (`m0` through `m3`), and 72 instructions.

The multiplier operates on two 16-bit source operands from either the `AR` registers or MAC16 (`MR`) registers.

Note: Each entry in the MAC16 `MR` register file has a unique assembly mnemonic of `m0` through `m3`.

Each operand may be taken from either the low or high half of a 32-bit register. The result of the operation is placed in the 40-bit accumulator. The `RSR`, `WSR`, and `XSR` instructions are used to read from and write to the `MR` registers and the `ACC` accumulator (either the lower 32 bits or the upper 8 bits). `MR[0]` and `MR[1]` can be used as the first multiplier input, and `MR[2]` and `MR[3]` can be used as the second multiplier input. Four of the 72 added instructions can overlap 32-bit `MR` register loads with multiply-accumulate operations.

The accumulator (`ACC`) and data registers (`MR`) are undefined after Reset.

2.10.1 MAC16 Architectural Additions

Table 21 and Table 22 show the MAC16 option's architectural additions.

Table 21. MAC16 Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W
ACC	1	40	Accumulator	R/W
m0	1	32	MAC16 Register 0 (`m0` in assembler, debugger, etc.)	R/W
m1	1	32	MAC16 Register 1 (`m1` in assembler, debugger, etc.)	R/W
m2	1	32	MAC16 Register 2 (`m2` in assembler, debugger, etc.)	R/W
m3	1	32	MAC16 Register 3 (`m3` in assembler, debugger, etc.)	R/W

Table 22. MAC16 Option Instruction Additions

Opcode[1]	Definition[2]
LDDEC	Load MAC16 data register (`MR`) with autodecrement
LDINC	Load MAC16 data register (`MR`) with autoincrement
UMUL.AA.**	Unsigned multiply of two address registers
MUL.AA.**	Signed multiply of two address registers
MUL.AD.**	Signed multiply of an address register and a MAC16 data register
MUL.DA.**	Signed multiply of a MAC16 data register and an address register
MUL.DD.**	Signed multiply of two MAC16 data registers
MULA.AA.**	Signed multiply-accumulate of two address registers
MULA.AD.**	Signed multiply-accumulate of an address register and a MAC16 data register
MULA.DA.**	Signed multiply-accumulate of a MAC16 data register and an address register
MULA.DD.**	Signed multiply-accumulate of two MAC16 data registers
MULS.AA.**	Signed multiply/subtract of two address registers
MULS.AD.**	Signed multiply/subtract of an address register and a MAC16 data register
MULS.DA.**	Signed multiply/subtract of a MAC16 data register and an address register
MULS.DD.**	Signed multiply/subtract of two MAC16 data registers
MULA.DA.**.LDDEC	Signed multiply-accumulate of a MAC16 data register and an address register, and Load a MAC16 data register with autodecrement
MULA.DA.**.LDINC	Signed multiply-accumulate of a MAC16 data register and an address register, and Load a MAC16 data register with autoincrement
MULA.DD.**.LDDEC	Signed multiply-accumulate of two MAC16 data registers, and Load a MAC16 data register with autodecrement
MULA.DD.**.LDINC	Signed multiply-accumulate of two MAC16 data registers, and Load a MAC16 data register with autoincrement

1. The * opcode parameter indicates whether the operands are taken from the Low or High 16-bit half of the AR or MR registers. The first * represents the location of the first operand; the second * represents the location of the second operand.
2. The destination for all product and accumulate results is the MAC16 accumulator.

2.10.2 MAC16 Use With the CLAMPS Instruction

The `CLAMPS` instruction, implemented with the Miscellaneous Operations Architectural Additions (see below), can be used in conjunction with the MAC16 option. This instruction can be used to clamp results to 16 bits before they are stored to memory.

2.11 32-bit Integer Multiply Option

This option provides instructions that implement 32-bit integer multiplication, providing single instruction targets for the multiplication operators of high-level programming languages such as C. Without this option, the Xtensa XCC compiler implement's 32-bit integer multiplication with subroutine calls, which use the 16-bit multiplier or multiplier/accumulator if available.

There is one sub-option within this option, called Mul32High, which controls whether the `MULSH` and `MULUH` instructions are included or not. For some implementations of the Xtensa LX processor, generating the high 32 bits of the product requires additional hardware, so disabling this sub-option may reduce cost.

2.11.1 32-bit Integer Multiply Architectural Additions

Table 23 and Table 24 show this option's architectural additions. This option adds no new processor state.

Table 23. 32-bit Integer Multiply Option Processor-Configuration Additions

Parameter	Description	Valid Values
`Mul32High`	Determines whether the `MULSH` and `MULUH` instructions are included	0 or 1
`MulAlgorithm`	Determines the multiplication algorithm employed	Implementation-dependent

Table 24. 32-bit Integer Multiply Instruction Additions

Instruction	Definition
MULL	Multiply Low (return least-significant 32 bits of product)
MULUH	Multiply Unsigned High (return most-significant 32 bits of product)
MULSH	Multiply Signed High (return most-significant 32 bits of product)

2.11.2 Sharing of Integer-Multiplier Hardware

If the MUL32 and MUL16 options are both selected, the MUL16 instruction will use the MUL32 option's 32-bit integer multiplier instead of adding another multiplier to the configuration. Similarly, the MAC16 and MUL16 options share the 16-bit multiplier/accumulator hardware block if both options are selected. However, the MAC16 option cannot share the MUL32 option's multiplier, so if both MAC16 and MUL32 options are selected, the configuration will have separate 16- and 32-bit multipliers. Table 25 lists all of the Xtensa LX configuration options that employ multipliers and illustrates how hardware is shared among them.

Table 25. Multiplier Hardware Sharing Among Xtensa LX Configuration Options

Configuration Option Selected?			Shared Multiplier?	Comments
MUL16	**MAC16**	**MUL32**		
Yes	No	No	No	Only 1 option selected, so no sharing
No	Yes	No	No	Only 1 option selected, so no sharing
No	No	Yes	No	Only 1 option selected, so no sharing
Yes	Yes	No	Yes	MUL16 and MAC16 options share a 16-bit multiplier
Yes	No	Yes	Yes	MUL16 and MUL32 options share a 32-bit multiplier
No	Yes	Yes	No	This configuration requires separate 16- and 32-bit multipliers
Yes	Yes	Yes	Yes	This configuration requires separate 16- and 32-bit multipliers

Also note that the floating-point unit (FPU) and Vectra LX configuration options add multipliers of their own and do not share any multiplier hardware with the MUL16, MAC16, or MUL32 configuration options.

2.12 Miscellaneous Operations Option

Miscellaneous instructions are enabled in groups. Each group provides the computational capabilities required by a specific set of applications.

2.12.1 Miscellaneous Operations Architectural Additions

Table 26 and Table 27 show the miscellaneous operations instruction-group options.

Table 26. Miscellaneous Operations Option Processor-Configuration Additions

Instruction Group	Description	Valid Values
NSA	Enable the normalization shift amount instructions: NSA, NSAU	0 or 1
MINMAX	Enable the minimum and maximum value instructions: MIN, MAX, MINU, MAXU	0 or 1
CLAMPS	Enable the signed clamp instruction: CLAMPS	0 or 1
SEXT	Enable the sign extend instruction: SEXT	0 or 1

Table 27. Miscellaneous Operations Instructions

Instruction	Definition
NSAU	Normalization Shift Amount Unsigned `AR[r] ← NSA(0, AR[s])` `NSAU` returns the number of contiguous bits in the most significant end of `AR[s]`, or 32 if `AR[s]`= 0. The result may be used as a left-shift amount such that the result of `SLL` on `AR[s]` will have bit 31 ≠ 0 (if `AR[s]`≠ 0).
NSA	Normalization Shift Amount signed $\texttt{AR[r]} \leftarrow \texttt{NSA(AR[s]}_{31}\texttt{, AR[s])}$ `NSA` returns the number of contiguous bits in the most significant end of `AR[s]` that are equal to the sign bit (not counting the sign bit itself), or 31 if `AR[s]` = 0 or -1. The result may be used as a left-shift amount such that the result of `SLL` on `AR[s]` will have bit 31 ≠ bit 30 (if `AR[s]`≠ 0).
MIN	Minimum Value Signed `AR[r] ← if AR[s] < AR[t] then AR[s] else AR[t]`
MAX	Maximum Value Signed `AR[r] ← if AR[s] < AR[t] then AR[t] else AR[s]`

MINU	Minimum Value Unsigned
	`AR[r] ← if (0‖AR[s]) < (0‖AR[t])`
	`        then AR[s]`
	`        else AR[t]`

MAXU	Maximum Value Unsigned
	`AR[r] ← if (0‖AR[s]) < (0‖AR[t])`
	`        then AR[t]`
	`        else AR[s]`

CLAMPS	Clamp to Signed Power of Two Range $\texttt{sign} \leftarrow \texttt{AR[s]}_{31}$ $\texttt{AR[r]} \leftarrow \texttt{if AR[s]}_{30..(t+7)} = \texttt{sign}^{24-t}$ `        then AR[s]` $\texttt{else sign}^{(25-t)} \parallel \texttt{(not sign)}^{t+7}$
SEXT	Sign Extend $\texttt{sign} \leftarrow \texttt{AR[s]}_{t+7}$ $\texttt{AR[r]} \leftarrow \texttt{sign}^{(23-t)} \parallel \texttt{AR[s]}_{t+7..0}$

2.13 Boolean Option

The Boolean option makes available a set of Boolean registers along with branches and other operations which refer to them. These Boolean architectural additions can be used by coprocessors such as the Tensilica-designed floating-point coprocessor, the Vectra LX DSP extension, and other designer-defined TIE language extensions.

2.13.1 Boolean Architectural Additions

Table 28 and Table 29 show this option's architectural additions.

Table 28. Boolean Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
BR	16	1	Boolean Registers	R/W	4

1. Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions.

Table 29. Boolean Option Instruction Additions

Instruction	Definition
BF	Branch if Boolean False
BT	Branch if Boolean True
MOVF	Conditional move if False
MOVT	Conditional move if True
ANDB	Boolean And
ANDBC	Boolean And with Complement
ORB	Boolean Or
ORBC	Boolean Or with Complement
XORB	Boolean Exclusive Or
ANY4	4-Boolean Or Reduction (result is 1 if any of the 4 booleans is true)
ALL4	4-Boolean And Reduction (result is 1 if all of the 4 booleans are true)
ANY8	8-Boolean Or Reduction (result is 1 if any of the 8 booleans is true)
ALL8	8-Boolean And Reduction (result is 1 if all of the 8 booleans are true)

2.13.2 Boolean Registers

The Boolean option provides 16 single-bit Boolean registers for storing the results of TIE-instruction and coprocessor comparison operations. The results of these operations can be subsequently tested by conditional-move and branch instructions. Note that the core Xtensa LX instructions do not use these optional Boolean registers (because they are optional).

Compared to condition codes used by other ISAs, these Boolean registers eliminate the common bottleneck of having only one place to store comparison results. It is possible, for example, to make multiple, iterative comparisons before using the comparison results. For Single-Instruction Multiple-Data (SIMD) operations, these Boolean registers provide storage for as many as 16 simultaneous comparison results and conditionals.

Instructions that produce Boolean values generate only one sense of the condition (for example, $=$ but not $\neq$), but all Boolean-register uses allow for complementing of the Boolean results, and conditional branches that use the Boolean results (`bt` and `bf`) can branch on the true or false result condition. Logical operations on the Boolean registers can replace spaghetti-branch code in many situations. One Boolean value can be computed from multiple Boolean results using the `ANY4`, `ALL4`, etc. instructions. This feature is useful, for example, for using one instruction to test the results of a SIMD comparison to determine if any or all of the elements satisfy a test such as testing if any byte of a word is zero.

Note: `ANY2` and `ALL2` instructions are implemented by the `ANDB` and `ORB` instructions using `bs+0` and `bs+1` as arguments.

The Boolean registers are undefined after Reset.

The Boolean registers are accessible from C using the `xtbool`, `xtbool2`, `xtbool4`, `xtbool8`, and `xtbool16` data types.

2.14 Coprocessor Option

The Xtensa LX coprocessor option allows for the grouping of processor state for the purpose of lazy context switching. A lazy context switch allows the program to defer the saving of coprocessor registers until those registers are used. Therefore, if the registers are not used by another context, they need not be saved. Multithreaded operating systems such as Wind River's VxWorks and ATI/Mentor Graphics' Nucleus must normally save and restore every processor register for each context switch. The Xtensa LX coprocessor option allows the designer to designate certain registers as coprocessor registers that need not always be saved and restored during context switches (some contexts will not use these registers, so the save and restore operations are not needed). Designating a coprocessor creates a bit in a special coprocessor enable register. When that bit is set, any access to the coprocessor's registers causes an exception. The associated exception-handling routine can then save the state of the coprocessor's registers before they're used, for example, in a new context.

The coprocessor option must be selected before specific coprocessors, such as the floating-point coprocessor option, can be added.

2.14.1 Coprocessor Option Architectural Additions

Table 30 shows the coprocessor option's architectural additions.

Table 30. Coprocessor Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
CPENABLE	1	8	Coprocessor Enable Bits	R/W	224

1. Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions.

2.14.2 Coprocessor Context Switch

The TIE compiler generates, for each coprocessor, the assembly code to save the state associated with a coprocessor to memory and to restore coprocessor state from memory using the TIE-generated `RUR` and `WUR` instructions. A series of `RUR` and `S32I` instructions saves the coprocessor state and a series of `L32I` and `WUR` instructions restores that state.

Note: The `RUR` and `WUR` instructions are used only for individual state registers and not for register files.

Coprocessors are enabled and disabled through the setting and clearing of bits in a `CPENABLE` register. The `CPENABLE` register allows "lazy" or conditional context switch of coprocessor state. Any instruction that references the state of coprocessor *n* (not including the shared Boolean registers) when that coprocessor's enable bit (bit n) is clear causes a `Coprocessor`*`n`*`Disabled` exception. `CPENABLE` can be cleared on a context switch. If the current task attempts to use the coprocessor thus generating a `Coprocessor`*`n`*`Disabled` exception, the exception handler can unload the previous task's coprocessor state and load the current task's. The appropriate `CPENABLE` bit is then set by the exception handler, which then returns to execute the coprocessor instruction.

Note: An `RSYNC` instruction must be executed after writing `CPENABLE` and before executing any instruction that references state controlled by the changed bits of `CPENABLE`.

The `CPENABLE` register is undefined after Reset.

2.14.3 System-Specific TIE Instructions

System-specific coprocessor instructions can be added using the TIE language. TIE instructions become part of the Xtensa LX instruction stream. The `LSCI` and `LSCX` intermediate opcodes are reserved for implementing subopcodes for coprocessor loads and stores.

2.14.4 Coprocessor Load/Store Addressing Modes

Coprocessors define their own set of loads to coprocessor registers and stores from those registers. The processor can form a coprocessor load/store address using any of the Xtensa LX addressing modes (listed in Table 31), but the following addressing modes are used automatically by the XCC compiler. (**Note:** x is a configurable per-instruction shift of 0 to 3.)

Table 31. Coprocessor Option Load/Store Addressing Modes

Name	Typical Mnemonic Suffix	Definition
Immediate	I	vAddr $\leftarrow$ AR[s] + $(0^{24-x}\|\|\text{imm8}\|\|0^{x})$
Immediate with Update	IU	vAddr $\leftarrow$ AR[s] + $(0^{24-x}\|\|\text{imm8}\|\|0^{x})$ AR[s] $\leftarrow$ vAddr
Indexed	X	vAddr $\leftarrow$ AR[s] + $(0^{32-x}\|\|\text{AR[t]}\|\|0^{x})$
Indexed with Update	XU	vAddr $\leftarrow$ AR[s] + $(0^{32-x}\|\|\text{AR[t]}\|\|0^{x})$ AR[s] $\leftarrow$ vAddr

2.15 Floating-Point Coprocessor Option

The floating-point coprocessor option is a Tensilica-designed coprocessor. It uses the prerequisite coprocessor and Boolean options and adds the logic and architectural components needed to execute IEEE754 single-precision floating-point operations. These operations are useful, for example, for DSP algorithms that require better than 16 bits of precision, such as high-quality audio compression and decompression. Also, DSP algorithms operating on less precise data are more easily coded using floating-point (because of the wide dynamic range), and floating-point operations boost the performance of many programs written in high-level programming in languages such as C. If the floating-point coprocessor option is not selected, the compiler emulates the floating-point operations in software.

2.15.1 Floating-Point State Additions

Table 32 summarizes the processor state added by the floating-point coprocessor. The FR register file consists of 16 registers of 32 bits each and is used for all data computation. Load and store instructions transfer data between the FR's and memory. The FV bits provide per-register valid bits similar to the AV's when the Speculation option is also enabled. The FCR has several fields that may be changed at run-time to control the operation of various instructions. The format of FCR is:

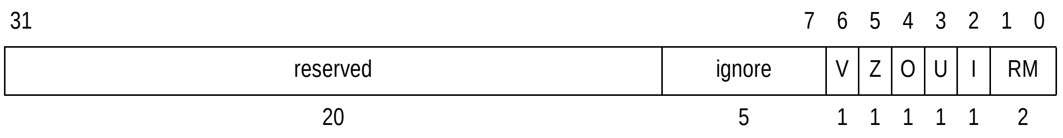


Table 32. FCR fields

FCR Field	Meaning
RM	Rounding Mode 0 ⟶ round to nearest 1 ⟶ round toward 0 (TRUNC) 2 ⟶ round toward $+\infty$ (CEIL) 3 ⟶ round toward $-\infty$ (FLOOR)
I	Inexact exception enable (0 ⟶ disabled, 1 ⟶ enabled)
U	Underflow exception enable (0 ⟶ disabled, 1 ⟶ enabled)
O	Overflow exception enable (0 ⟶ disabled, 1 ⟶ enabled)
Z	Divide-by-zero exception enable (0 ⟶ disabled, 1 ⟶ enabled)
V	Invalid exception enable (0 ⟶ disabled, 1 ⟶ enabled)
ignore	Reads as 0, ignored on write
reserved	Reads back last value written. Non-zero values cause a Floating Point Exception on any floating-point operation.

The FSR (fields listed in Table 33) provides the status flags required by the IEEE-754 standard. Any operation that raises a non-enabled exception sets these flags. Enabled exceptions abort the operation with a floating-point exception and do not set the flags.

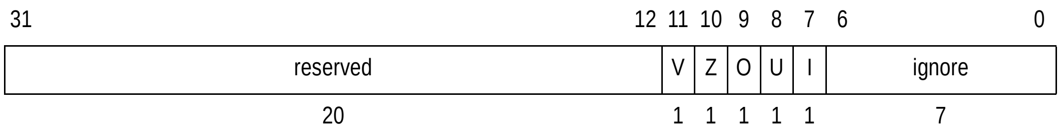


Table 33. FSR fields

FSR Field	Meaning
I	Inexact exception flag
U	Underflow exception flag
O	Overflow exception flag
Z	Divide-by-zero flag

Table 33. FSR fields (continued)

FSR Field	Meaning
V	Invalid exception flag
ignore	Reads as 0, ignored on write
reserved	Reads back last value written. Non-zero values cause a Floating Point Exception on any floating-point operation.

The Xtensa LX processor's FCR may be read and written without waiting for the results of pending floating-point operations. Writes to FCR affect subsequent floating point operations, but there is usually little performance degradation from this dependency. Only reads of FSR, because FSR depends on all previous floating-point operations, need cause a significant pipeline interlock.

2.15.2 Floating-Point Instructions and Data Types

Table 34 shows the floating-point instructions, operations, and associated data types.

Table 34. Floating-Point Instructions, Operations, and Data Types

Instruction	Operation and Data Type
LSI	Load Single-Precision Immediate
LSIU	Load Single-Precision Immediate with Base Update
SSI	Store Single-Precision Immediate
SSIU	Store Single-Precision Immediate with Base Update
LSX	Load Single-Precision
LSXU	Load Single-Precision with Base Update
SSX	Store Single-Precision
SSXU	Store Single-Precision with Base Update
ADD.S	Single-Precision Add
SUB.S	Single-Precision Subtract
MUL.S	Single-Precision Multiply
MADD.S	Single-Precision Multiply-Add
MSUB.S	Single-Precision Multiply-Subtract
FLOAT.S	Single-Precision Signed Integer to Floating-Point Conversion
UFLOAT.S	Single-Precision Unsigned Integer to Floating-Point Conversion
ROUND.S	Single-Precision Floating-Point to Signed Integer Conversion with Round to Nearest
TRUNC.S	Single-Precision Signed Integer to Floating-Point Conversion with Round to 0
UTRUNC.S	Single-Precision Unsigned Integer to Floating-Point Conversion with Round to 0
FLOOR.S	Single-Precision Signed Integer to Floating-Point Conversion with Round to -∞

Table 34. Floating-Point Instructions, Operations, and Data Types (continued)

Instruction	Operation and Data Type
`CEIL.S`	Single-Precision Signed Integer to Floating-Point Conversion with Round to +∞
`MOV.S`	Single-Precision Move
`NEG.S`	Single-Precision Negate
`ABS.S`	Single-Precision Absolute Value
`MOVEQZ.S`	Single-Precision Move If Equal to Zero
`MOVNEZ.S`	Single-Precision Move If Not Equal to Zero
`MOVLTZ.S`	Single-Precision Move If less than Zero
`MOVGEZ.S`	Single-Precision Move If Greater than or Equal to Zero
`MOVF.S`	Single-Precision Move If Boolean Condition False
`MOVT.S`	Single-Precision Move If Boolean Condition True
cond`.S`	Single-Precision Compare with Boolean result
`RFR`	Read Floating-Point Register (FR to AR)
`WFR`	Write Floating-Point Register (AR to FR)
`RFCR`	Read Floating-Point Control Register (to AR)
`WFCR`	Write Floating-Point Control Register (from AR)

The primary floating-point data type is IEEE-754 single-precision, as shown in Figure 13.

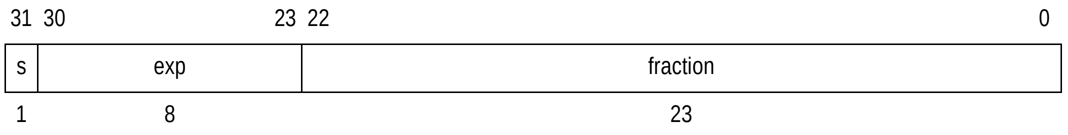


Figure 13. Single-Precision Floating-Point Representation Format

The other floating-point data format is a signed, 32-bit integer used by the `FLOAT.S`, `TRUNC.S`, `ROUND.S`, `FLOOR.S`, and `CEIL.S` instructions.

The IEEE-754 floating-point standard uses a sign-magnitude format, with a 1-bit sign, an 8-bit exponent with bias 127, and a 24-bit significand formed from 23 explicit, stored bits representing the binary digits to the right of the binary point, and an implicit bit to the left of the binary point (0 if exponent is zero, 1 if exponent is non-zero). Thus, the value of the number is:

$$(-1)^{s} \times 2^{\mathtt{exp}-127} \times \mathtt{implicit.fraction}$$

Thus, the representation for 1.0 is 0x3F800000, with a sign of 0, exp of 127, a zero fraction, and an implicit 1 to the left of the binary point.

The Xtensa ISA includes IEEE-754 signed-zero, infinity, quiet NaN, and sub-normal representations and processing rules. The ISA does not include IEEE-754 signaling NaNs or exceptions. Conversions between integer and floating-point representations use a binary scale factor to make conversion into and out of fixed-point formats faster.

2.15.3 Floating-Point Exceptions

Xtensa Floating-Point exceptions are caused by the execution of a floating-point instruction opcode not implemented by the processor hardware, or by an operation executed in hardware that attempts to raise an enabled (as determined by the FCR) IEEE754 exception. Disabled IEEE-754 exceptions simply set the corresponding flag bit in the FSR and write the IEEE-754-defined value into the result register.

On implementations with precise exceptions, the Floating-Point Exception aborts the instruction raising the condition and no Floating-Point state (FR or FSR) is modified. When imprecise exceptions are in use, FR and FSR are written as if the exception were not enabled, and the exception occurs on a later instruction.

Software must look at the instruction that caused the exception and its operands to determine the cause of a floating-point exception. There is no floating-point equivalent of XCCAUSE.

2.15.4 Floating-Point Instructions

The Xtensa LX floating-point instructions are defined in Table 35 and Table 36. The instructions operate on data in the floating-point register file, which consists of 16 32-bit registers.

Table 35. Floating-Point Coprocessor Option Load/Store Instructions

Instruction	Definition

LSI	Load Single-Precision Immediate
	vAddr ← AR[s] + (0^{22}‖imm8‖0^2)
	FR[t] ← Load32(vAddr)

LSIU	Load Single-Precision Immediate with Base Update
	vAddr ← AR[s] + (0^{22}‖imm8‖0^2)
	FR[t] ← Load32(vAddr)
	AR[s] ← vAddr

SSI	Store Single-Precision Immediate vAddr ← AR[s] + (0^{22}‖imm8‖0^2) Store32 (vAddr, FR[t])

Table 35. Floating-Point Coprocessor Option Load/Store Instructions (continued)

Instruction	Definition

SSIU	Store Single-Precision Immediate with Base Update
	`vAddr` $\leftarrow$ `AR[s] + (`0^{22}`‖imm8‖`0^{2}`)`
	`Store32 (vAddr, FR[t])`
	`AR[s]` $\leftarrow$ `vAddr`

LSX	Load Single-Precision
	`vAddr` $\leftarrow$ `AR[s] + AR[t]`
	`FR[t]` $\leftarrow$ `Load32(vAddr)`

LSXU	Load Single-Precision with Base Update
	`vAddr` $\leftarrow$ `AR[s] + AR[t]`
	`FR[t]` $\leftarrow$ `Load32(vAddr)`
	`AR[s]` $\leftarrow$ `vAddr`

SSX	Store Single-Precision
	`vAddr` $\leftarrow$ `AR[s] + AR[t]`
	`Store32 (vAddr, FR[r])`

SSXU	Store Single-Precision with Base Update
	`vAddr` $\leftarrow$ `AR[s] + AR[t]`
	`Store32 (vAddr, FR[r])`
	`AR[s]` $\leftarrow$ `vAddr`

Table 36. Floating-Point Coprocessor Option Operation Instructions

Instruction	Definition
ADD.S	Single-Precision Add $FR[r] \leftarrow FR[s] +_s FR[t]$
SUB.S	Single-Precision Subtract $FR[r] \leftarrow FR[s] -_s FR[t]$
MUL.S	Single-Precision Multiply $FR[r] \leftarrow FR[s] \times_s FR[t]$
MADD.S	Single-Precision Multiply/Add $FR[r] \leftarrow FR[r] +_s (FR[s] \times_s FR[t])$
MSUB.S	Single-Precision Multiply/Subtract $FR[r] \leftarrow FR[r] -_s (FR[s] \times_s FR[t])$
FLOAT.S	Convert Integer to Single-Precision and scale $FR[r] \leftarrow float_s(FR[s]) \times_s 2.0^{t-16}$
ROUND.S	Scale and Convert Single-Precision to Integer, round to nearest $FR[r] \leftarrow round_s(FR[s] \times_s 2.0^{t+16})$
TRUNC.S	Scale and Convert Single-Precision to Integer, round to 0 $FR[r] \leftarrow trunc_s(FR[s] \times_s 2.0^{t+16})$
FLOOR.S	Scale and Convert Single-Precision to Integer, round to $-\infty$ $FR[r] \leftarrow floor_s(FR[s] \times_s 2.0^{t+16})$

Table 36. Floating-Point Coprocessor Option Operation Instructions (continued)

Instruction	Definition
`CEIL.S`	Scale and Convert Single-Precision to Integer, round to $+\infty$ $\texttt{FR[r]} \leftarrow \texttt{ceil}_S(\texttt{FR[s]} \times_S \texttt{2.0}^{\texttt{t-16}})$
`MOV.S`	Single-Precision Move $\texttt{FR[r]} \leftarrow \texttt{FR[s]}$
`ABS.S`	Single-Precision Absolute Value $\texttt{FR[r]} \leftarrow \texttt{abs}_S(\texttt{FR[s]})$
`MOVEQZ.S`	Single-Precision Conditional Move if Equal to Zero $\texttt{if AR[t] = } 0^{32} \texttt{ then FR[r]} \leftarrow \texttt{FR[s] endif}$
`MOVNEZ.S`	Single-Precision Conditional Move if Not Equal to Zero $\texttt{if AR[t]} \neq 0^{32} \texttt{ then FR[r]} \leftarrow \texttt{FR[s] endif}$
`MOVLTZ.S`	Single-Precision Conditional Move if less than Zero $\texttt{if AR[t]}_{31} < 0 \texttt{ then FR[r]} \leftarrow \texttt{FR[s] endif}$
`MOVGEZ.S`	Single-Precision Conditional Move if Greater than or Equal to Zero $\texttt{if AR[t]}_{31} \texttt{ >= 0 then FR[r]} \leftarrow \texttt{FR[s] endif}$
`MOVF.S`	Single-Precision Conditional Move if False $\texttt{if BR}_t \texttt{ = 0 then FR[r]} \leftarrow \texttt{FR[s] endif}$
`MOVT.S`	Single-Precision Conditional Move if True $\texttt{if BR}_t \neq \texttt{0 then FR[r]} \leftarrow \texttt{FR[s] endif}$
cond`.S`	Single-Precision Compare $\texttt{BR}_r \leftarrow \texttt{FR[s]}\ cond_S\ \texttt{FR[t]}$ *cond* = `unordered, ordered and equal, unequal, ordered and less than, unordered and less than, ordered and less than or equal, unordered and less than or equal.`
`RFR`	Move from FR to AR $\texttt{AR[r]} \leftarrow \texttt{FR[s]}$
`WFR`	Move from AR to FR $\texttt{FR[r]} \leftarrow \texttt{AR[s]}$

2.16 Vectra LX Vector DSP Engine Option

The Vectra LX option adds an entire FLIX-based (Flexible Length Instruction Xtensions) vector DSP engine to the Xtensa LX architecture. The basic Xtensa LX architecture, implements a single-issue processor with 24/16-bit instruction encoding and one load/store unit. (A second load/store unit is a configuration option for the Xtensa LX processor.) An Xtensa LX processor configured with the Vectra LX DSP engine option is also a single-issue machine, but Vectra LX instructions are encoded in 64 bits. These instructions can be freely intermixed with the processor's 24- and 16-bit instructions. Each Vectra LX instruction contains three independent operations.

The Vectra LX configuration option uses both the standard and optional Xtensa LX load/store units. This dual-load/store ability allows the Xtensa LX with the Vectra LX DSP engine to perform XY memory operations, permitting very high performance execution of many DSP algorithms.

The Vectra LX instruction-word format (little-endian version) appears in Figure 14 below.

63 46	45 28	27 4	3			0
ALU/2nd LDST	MAC	Xtensa LX/Vectra LX LDST	1	1	1	0
18	18	24	4			

Figure 14. Vectra LX Engine Instruction Word Format (Little-Endian Configuration)

The right-most four bits of the Vectra LX instruction format take the decimal value 14. These four bits in an Xtensa instruction word determine the instruction's length and format. Tensilica has defined the Vectra LX instructions within the Xtensa LX processor's FLIX framework using the 4-bit value of 14. The remaining bits of the 64-bit instruction word are interpreted as follows:

- Bits [27:4] define the first Vectra LX instruction slot, which contains either an Xtensa LX core instruction or a Vectra LX load or store instruction.
- Bits [45:28] define the second Vectra LX instruction slot, which contains either a multiply-accumulate (MAC) instruction or a select instruction.
- Bits [63:46] define the third Vectra LX instruction slot, which contains ALU and shift instructions or a load or store instruction for the second Vectra LX load/store unit.

2.16.1 Vectra LX State

The Vectra LX state consists of four state registers and three register files as summarized below in Table 37.

The `VSAR` state register is a 5-bit register, similar to the Xtensa core `SAR` shift amount register in that it specifies the shift amount for variable shift operations. This state register is used in conjunction with the shift and "pack" instructions of Vectra LX.

The `ROUND` state register stores a rounding value typically added to a data element stored in a high-precision format before it is converted into a lower precision format. This state register is used in conjunction with the Vectra LX engine's multiply accumulate instructions.

The `CBEGIN` and `CEND` store the beginning and ending addresses of a memory-resident circular buffer. These registers are used in conjunction with special Vectra LX load and store instructions that provide automatic address wrap-around for the circular buffer.

The Vectra LX engine's vec general-purpose register file consists of sixteen 160-bit registers that hold the operands for all Vectra LX data processing. Vectra LX load and store instructions load data into these registers and store the contents of these registers into memory.

The valign register file is a special-purpose register file that accelerates the loading or storing of unaligned data to or from the general-purpose vec register file. There are four 128-bit registers in the valign register file. Of the 128 bits in each valign register, 112 bits are used to store data for unaligned load/store operations. One flag bit indicates the status of the alignment register for unaligned store operations. While architecturally the flag needs to be a single bit, it is implemented as the remaining 16 bits of the alignment register for ease of coding.

The vsel register file is a special-purpose register file used in conjunction with the select (SEL) instruction. This instruction rearranges data from the source vector registers into the destination vector register. The select register file specifies the selection control values for this rearrangement. There are eight 32-bit registers in this file.

Table 37. Vectra LX Engine State

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W
VSAR	1	5	Vector Shift Amount Register	R/W
ROUND	1	40	Rounding Value	R/W
CBEGIN, CEND	1 each	32	Virtual Address Registers for Circular-Buffer Support	R/W
V (vec)	16	160	General-Purpose Vectra LX Register File	R/W
U (valign)	4	128	Register File to Accelerate Unaligned Loads and Stores	R/W
S (vsel)	8	32	Vector SEL control values	R/W

2.16.2 Vectra LX Data Types

The Vectra LX DSP engine processes fixed-point data. The basic data element is 16 bits wide and a vector consists of 8 such 16-bit elements. Thus, an input vector in memory is 128-bits wide. The Vectra LX engine also supports a wider 32-bit data element with four such elements in a 128-bit vector. This double-width data type is typically generated as the result of multiply or multiply/accumulate operations on 16-bit data elements.

Note: Selecting the Vectra LX option in an Xtensa LX configuration automatically selects the 128-bit PIF (processor interface) option.

When a 128-bit vector is loaded from memory into a Vectra LX register, the Vectra LX engine automatically expands the vector to 160-bits by adding guard bits. The extra guard bits provide higher precision for intermediate computations. During a load operation, the Vectra LX engine expands 16-bit scalar values to 20 bits and it expands 32-bit scalar values to 40-bits. The Vectra LX engine supports both signed and unsigned data types and provides a symmetrical set of load and store instructions for both these data types. For signed data, a load results in sign extension from 16 to 20 (or 32 to 40) bits. For unsigned data, loads result in zero extension.

During memory stores, the Vectra LX automatically compresses the expanded data elements back to their original size. Vectra LX store operations result in saturation of data if the data-element value held in the Vectra LX register exceeds the capacity of the data element's width as stored in memory (16 or 32 bits).

Because of the presence of guard bits and the semantics of the Vectra LX load and store instructions, the Vectra LX data-processing instructions (such as the ALU and MAC instructions) come in only one variety – they treat the data as signed. This convention allows the Vectra LX to correctly process unsigned data as long as there is no overflow into the most significant bit of the elements in a vector register. Software must ensure that the input data is scaled appropriately to prevent such an overflow when processing unsigned numbers. Note that this problem is no different than the general problem of preventing overflows when running code on any fixed-point processor.

2.16.3 Vectra LX Load and Store Instructions

The Vectra LX engine provides a rich set of load and store instructions for loading both vector and scalar operands. These instructions support a number of different addressing modes and provide great flexibility to the programmer. The different types of load/store instructions (along with their addressing mode) are briefly described in this section.

Vectra LX Load and Store Addressing Modes

All Vectra LX addressing modes involve a base address and an offset. The base address is always specified by a value stored in an Xtensa LX AR register. The offset can be either an immediate value or specified by a value in another AR register. Immediate offsets are typically biased to provide a small negative and a large positive range. When an AR register provides the offset value, it is treated as a signed, 32-bit offset. All Vectra LX address calculations pre-increment the base address with the offset.

Note: Because the address offset can be negative, the actual address may be incremented or decremented. For simplicity in this explanation, we use the term increment to refer to both.

For some instructions, the address offset equals the vector size (unit stride), so it is not encoded as part of the instruction. This is true for the priming and flush instructions that load and store unaligned vectors from and to memory. It is also true for the circular-buffer load and store instructions. Table 38 summarizes the addressing modes and shows the notation used to refer to them.

Table 38. Vectra LX Addressing Modes

Name	Notation	Description
`Immediate`	`.I`	Offset specified as a biased immediate operand
`Immediate with Update`	`.IU`	Offset specified as a biased immediate operand. Base address updated with the incremented address.
`Index`	`.X`	Offset specified as a signed, 32-bit value from an AR register.
`Index with Update`	`.XU`	Offset specified as a signed, 32-bit value from an AR register. Base address updated with the incremented address.

Aligned Vector Load and Store Instructions

The aligned vector load and store instructions move 128-bit vectors between the Vectra LX registers and memory addresses aligned to 128-bit boundaries. The least significant 4 bits of the byte address are ignored when generating the physical memory address. These instructions support all four addressing modes.

Unaligned Vector Load and Store Instructions

The unaligned vector load and store instructions move 128-bit vectors between the Vectra LX registers and memory addresses that may or may not be aligned to 128-bit boundaries. If the address is aligned, these instructions perform the same operation as aligned load and store instructions. For unaligned addresses, these instructions use the Vectra LX alignment register file to provide a throughput of one unaligned load or store operation per instruction. The unaligned vector load and store instructions rely on two mechanisms to do this. One mechanism is the rotation of load and store data based on the least significant address bits of the virtual address. The other mechanism is the appropriate merging of the vector data with the contents of the alignment register.

A special priming instruction is used to begin the process of loading an array of unaligned data. This instruction conditionally loads the alignment register if the target address is unaligned. If the memory address is not aligned to a 128-bit boundary, this load initializes the contents of the alignment register. The subsequent unaligned load instruction pre-increments the load target address and then merges data loaded from the target location with the appropriate data bytes already residing in the alignment register to form the completed vector, which is then written to the vector register. Data from this load then overwrites the alignment register, priming it for the next load. Subsequent load instructions provide a throughput of one unaligned load per instruction.

The design of the priming load and unaligned load instructions is such that they can be used in situations where the alignment of the address is unknown. If the address is aligned to a 128-bit boundary, the priming load instruction does nothing. Subsequent unaligned load instructions will not use the alignment register and will directly load the memory data into the vector register. Thus, the load sequence works whether the starting address is aligned or not.

Unaligned stores operate in a slightly different manner. Each unaligned store instruction is sensitive to the value of the flag bit in the alignment register. If the flag bit is 1, appropriate bytes of the alignment register are combined with appropriate bytes of the vector register to form the 128-bit store data written to memory. On the other hand, if the flag bit is 0, then the store is a partial store and only the relevant bytes of the vector register are written to memory. Data from the alignment register is not used. No data will be written to one or more bytes starting at the 128-bit aligned address. This store will only write data starting from the byte corresponding to the memory address of the store.

Each unaligned store instruction (independent of the value of the flag bit) will also update the appropriate bytes in the alignment register, priming it for the next store instruction. Every unaligned store instruction also sets the alignment register's flag bit to 1. When the last unaligned store instruction executes, some data may be left in the alignment register, which must be flushed to memory. A special flush instruction copies this data from the alignment register to memory if needed.

Start with the `ZALIGN` instruction to store an array of vectors beginning at an unaligned memory address. This instruction initializes the alignment register's contents and clears the flag bit. A series of unaligned stores following the `ZALIGN` instruction will store one vector to memory per instruction. Note that the first store instruction of this series will perform a partial store because the flag bit was cleared by the `ZALIGN` instruction. Each subsequent store will perform a full 128-bit store because the first (and subsequent) store instructions set the flag bit. Finally, a flush instruction flushes out the last remaining bytes in the alignment register.

Once again, the design of the unaligned store and flush instructions allows them to work even if the memory address is aligned to a 128-bit boundary. Specifically, if the address is aligned, the store instructions store data only from the vector register. The alignment register is not used if the addresses are aligned. Similarly, if the addresses are aligned, the flush instruction does nothing.

Note that these instructions move data between memory and the Vectra LX registers if the memory addresses are not aligned to a vector boundary. However, these instructions do assume that the addresses are aligned to 16-bit scalar boundaries. If these instructions are used with addresses that are not aligned to 16-bit boundaries, the result is undefined. There are also 32-bit versions of these store and flush instructions that assume 32-bit scalar alignment.

The priming and flush instructions support only the immediate addressing mode, while the other instructions support all four addressing modes.

Circular Buffer Load and Store Instructions

The Vectra LX engine provides a special set of load and store instructions to operate a circular memory buffer. These instructions only support a unit-stride-offset-with-update addressing mode. The base address is “wrapped around” to the beginning of the buffer whenever the incremented address goes past the end of the circular buffer. Circular-buffer operations are supported for both aligned and unaligned vectors in memory.

The operation for a circular buffer load can be defined as follows:

```
ReadData = Memory[BasePointer + 16];
        if (BasePointer >= CEND) {
             BasePointer = BasePointer – CEND + CBEGIN;
        } else {
             BasePointer = BasePointer + 16;
        }
```

Note from the above code that the base pointer is always pre-incremented by 16 bytes (a unit stride) to form the data load address. However, `CEND` is compared to the base pointer, not to the incremented value. This detail prevents a critical timing path in the address-update logic. Consequently, the state register `CEND` must be initialized to a value that is 32 bytes less than the data buffer’s end address. Similarly, the state register `CBEGIN` must be initialized to a value that is 16 bytes less than the data buffer’s start address. Circular-buffer stores operate in a similar manner and the same rules and considerations apply to their operation.

Scalar Vectra LX Load and Store Instructions

Scalar loads fetch a scalar data element from memory and replicate it into a vector register. Scalar store instructions write the right-most element of a vector register into memory. Scalar load and store instructions support all four Vectra LX addressing modes.

The Vectra LX provides a special instruction to reduce the number of scalar loads executed by certain DSP kernels. This instruction, `REP20`, replicates any one of the eight 20-bit scalar elements (selected by a 3-bit immediate operand) in the input vector register into all eight scalar elements in the output vector register. Vectra LX also has a `REP40` instruction that replicates a selected 40-bit scalar element four times in the output vector register.

Special Vectra LX Load and Store Instructions

A special set of load and store instructions is provided to save and restore the Vectra LX vector registers to and from memory. The XCC compiler uses these instructions to save and restore Vectra LX registers when managing register allocation. Further, the compiler saves and restores the 160-bit-wide vector register file without extension/saturation.

Also included in this special class are instructions to load and store the Vectra LX alignment and select registers.

Second Load/Store Unit

The Vectra LX engine incorporates two load/store units that can be used simultaneously. The “primary” load/store unit is associated with bits [27:4] of the Vectra LX instruction word (the Xtensa/Vectra `LDST` instruction slot as depicted in Figure 14). This load/store unit is referred to as the primary load/store unit because it supports all the Vectra LX load and store instructions in addition to the core Xtensa LX load and store instructions.

The second Vectra LX load/store unit is associated with bits [63:46] of the instruction word (the ALU/2nd `LDST` instruction slot as depicted in Figure 19). This instruction slot only supports eight of the most commonly used Vectra LX load and store instructions. Further, this second load/store unit does not support any of the core Xtensa LX load or store instructions.

The primary reason to restrict the number of load/store instructions available to the second Vectra LX load/store unit is the limited opcode space available in this slot. Of the eight instructions available in this slot, four are load instructions and the other four are store instructions. These include updating and non-updating vector loads and stores, scalar loads and stores, and circular-buffer loads and stores.

2.16.4 Vectra LX MAC Instructions

The Vectra LX engine supports *real* and *complex* multiplication. The Vectra LX MAC instructions include four different accumulation options: no accumulation (multiply only), multiply/add, multiply/subtract, and multiply and add `ROUND` register. The MAC hardware consists of four 18-bit multipliers, which each generate a 36-bit product that is accumulated in a 40-bit accumulator.

Real-number multiplication results in the multiplication of corresponding elements of the input-vector operands. Because there are four hardware multipliers, a pair of multiply instructions is needed to multiply all eight input operands. The `MUL.0` instructions multiply the even elements of the input vectors and the `MUL.1` instructions multiply the odd elements of the input vectors. This 2-step process splits the even and odd products into separate registers.

Complex-number multiplication involves the multiplication of the real part and the imaginary part of the operands. The Vectra LX engine provides an instruction to accumulate the real part of a complex product into a vector of real elements and another instruction to accumulate the imaginary part of a complex product into a vector of imaginary elements.

The Vectra LX complex-number multiplication instructions operate in an iterative fashion, performing eight multiplications using the four hardware multipliers over three clock cycles.

Note: the Vectra LX multipliers are fully pipelined, so a new complex-number multiplication operation can be initiated every two clock cycles.

The complex-number-multiplication instructions do not have "`.0`" and "`.1`" versions. It is possible to use the iterative instruction feature for the complex-multiplication instructions because the eight multiplication operations produce only four 40-bit values, which can be stored in one vector register. For the real-multiplication instructions, performing eight multiplications using the iterative multiply feature does not help because the result produces eight 40-bit values, which require two vector registers to store the result. Thus, the real-number multiplication instructions are split into the odd and even pairs.

Both the real- and complex-number multiplication instructions result in the accumulator results being de-interleaved into two separate registers. The `PACK` instruction, described later, interleaves this data back into its original format.

2.16.5 Vectra LX ALU and Shift Instructions

The Vectra LX engine provides a number of different ALU and shift instructions. Most of these instructions come in two varieties – one that operates on 20-bit narrow registers and the other that operates on 40-bit wide registers.

Arithmetic Instructions

Add, subtract, minimum, and maximum are the three types of three-operand arithmetic instructions supported. Two-operand arithmetic instructions that generate the absolute value and the negated value of the input vector are also supported.

Logical Instructions

The Vectra LX instruction set contains logical AND, NAND, OR, and Exclusive-OR operations. Because these instructions operate on a bit-wise basis (as opposed to a scalar- element basis), each of these logical instructions only has one version.

Compare Instructions

The Vectra LX compare instructions compare the contents of two vector registers and update the Xtensa LX Boolean registers based on the comparison result. The operators provided are “less than,” “less than or equal,” and “equal to.”

Special ALU Instructions

The Vectra LX engine provides special ALU instructions to accelerate the performance of many DSP kernels on a vector processor. One such operation is reduction add, which adds all the elements of an input-vector operand to produce a scalar sum. The scalar sum is then replicated into all the elements of the result vector. Another set of instructions is an extension of the minimum and maximum instructions mentioned above. In addition to placing the minimum (or maximum) values into the result vector register, this instruction also sets the Xtensa LX Boolean registers to indicate which of the (two) input-vector elements was selected.

The “multiply sign” instruction is another special Vectra LX instruction. It is modeled after the “copy sign” operation used in floating-point data manipulations. The scalar operation corresponding to this instruction is illustrated by the following pseudo-code:

```
if (input0 > 0) { // input operand 0 is positive
          result = input1;
} else if (input0 < 0) { // input operand 0 is negative
          result = -input1;
} else { // input operand 0 is zero
          result = 0;
}
```

Vectra LX Shift Instructions

The Vectra LX shift instructions come in two formats: one specifies the shift amount as a 5-bit immediate operand and the other uses the state register VSAR to specify the shift amount. This class of instructions performs left-shift and arithmetic right-shift operations.

A special saturating left-shift instruction is also included in the Vectra LX instruction set. The shift amount for this instruction is specified through the VSAR register (an immediate shift amount is not available for this instruction).

2.16.6 Other Vectra LX Instructions

This section lists a few other instructions that do not specifically belong to the Load/Store, ALU, or MAC class of instructions.

algorithms. Its presence allows the programmer to issue a MAC and select instruction in the same instruction packet, which eases the coding of many algorithms. Vectra LX also has an `ESHFT40` instruction for shifting 40-bit elements.

Pack Instruction

The `PACK` instruction converts the accumulator contents back to the original, 20-bit-scalar data format. This instruction operates on two vector registers, assumes that each contains four 40-bit scalar values, converts these values, and places them into one vector register containing eight 20-bit scalar values. The 40-bit data format is typically generated as a result of multiply-accumulate operations on 20-bit input data. Note also that the multiplication instructions de-interleave data into two accumulator registers as explained earlier. The `PACK` instruction shifts each 40-bit element to the right by the amount specified in the `VSAR` register. It then saturates this shifted value to 20-bits, interleaves the data from the two input registers, and finally writes this value to the result register.

2.16.7 Vectra LX Instruction List – Grouped by Function

The following tables list all Vectra LX instructions, grouped by their functional class. The instructions are listed in assembly language format, along with comments describing their function. The assembly-language instruction format uses the notation shown in Table 40.

Table 40. Vectra LX Assembly-Language Instruction Notation

Designation	Description
vr, vs, vt	General purpose Vectra II registers indexed by the r, s, and t instruction fields respectively.
ars, art	Xtensa core address registers indexed by the s and t fields respectively.
ba8, ba4	Xtensa core Boolean registers indexed by the a8 and a4 fields respectively.
bb8, bb4	Xtensa core Boolean registers indexed by the b8 and b4 fields respectively.
br8, br4	Xtensa core Boolean registers indexed by the r field.
bs8, bs4	Xtensa core Boolean registers indexed by the s field.
bt8, bt4	Xtensa core Boolean registers indexed by the t field.
imm<n>	An n-bit immediate field.
uu	Alignment register indexed by the u field.
uur	Alignment register indexed by the ur field.
uut	Alignment register indexed by the ut field.
se	Select register indexed by the e field.

Table 40. Vectra LX Assembly-Language Instruction Notation (continued)

Designation	Description
ser	Select register indexed by the er field.
set	Select register indexed by the et field.
repsel	Immediate field for selecting the element to be replicated.

2.16.8 Vectra LX Load Instructions

Most Vectra LX load instructions have a signed and an unsigned version, illustrated in Table 41 and Table 42 by the syntax `LV{S,U}16` where `LVS16` refers to a signed load and `LVU16` refers to an unsigned load. Note that the priming and flush instructions for unaligned loads and the special load instructions do not distinguish between signed and unsigned operations.

Table 41. Vectra LX Aligned Vector Load Instructions

Instruction	Operands	Description
LV{S,U}16.I	vr, ars, imm8	Load Vector (base + immediate)
LV{S,U}32.I	vr, ars, imm8	
LV{S,U}16.IU	vr, ars, imm8	Load Vector (base + immediate with update)
LV{S,U}32.IU	vr, ars, imm8	
LV{S,U}16.X	vr, ars, art	Load Vector (base + index)
LV{S,U}32.X	vr, ars, art	
LV{S,U}16.XU	vr, ars, art	Load Vector (base + index with update)
LV{S,U}32.XU	vr, ars, art	
LV{S,U}16.CU	vr, ars	Load Vector Circular, unit stride with update
LV{S,U}32.CU	vr, ars	

Table 42. Vectra LX Unaligned Vector Load Instructions

Instruction	Operands	Description
LVA.P	uu, ars	Prime Align for vector load
LV{S,U}16A.I	vr, uu, ars, imm4	Load Vector Align (base + immediate)
LV{S,U}32A.I	vr, uu, ars, imm4	
LV{S,U}16A.IU	vr, uu, ars, imm4	Load Vector Align (base + immediate with update)
LV{S,U}32A.IU	vr, uu, ars, imm4	
LV{S,U}16A.X	vr, uu, ars, art	Load Vector Align (base + index)

Table 42. Vectra LX Unaligned Vector Load Instructions (continued)

Instruction	Operands	Description
LV{S,U}32A.X	vr, uu, ars, art	
LV{S,U}16A.XU	vr, uu, ars, art	Load Vector (base + index with update)
LV{S,U}32A.XU	vr, uu, ars, art	
LVA.CP	uu, ars	Prime Align for Circular Vector Load
LV{S,U}16A.CU	vr, uu, ars	Load Vector Align Circular, unit stride with update
LV{S,U}32A.CU	vr, uu, ars	

Table 43. Vectra LX Scalar Load Instructions

Instruction	Operands	Description
LS{S,U}16.I	vr, ars, imm8	Load Scalar (base + immediate)
LS{S,U}32.I	vr, ars, imm8	
LS{S,U}16.IU	vr, ars, imm8	Load Scalar (base + immediate with update)
LS{S,U}32.IU	vr, ars, imm8	
LS{S,U}16.X	vr, ars, art	Load Scalar (base + index)
LS{S,U}32.X	vr, ars, art	
LS{S,U}16.XU	vr, ars, art	Load Scalar (base + index with update)
LS{S,U}32.XU	vr, ars, art	

Table 44. Vectra LX Special Load Instructions

Instruction	Operands	Description
LVH.I	vr, ars, imm10	Load Vector High (base + immediate)
LALIGN.I	uu, ars, imm8	Load Align Register (base + immediate)
LSEL.I	se, ars, imm8	Load Select Register (base + immediate)

2.16.9 Vectra LX Store Instructions

Most Vectra LX store instructions (Tables 45 - 48) have a signed and an unsigned version, illustrated below by the syntax `SV{S,U}16` where `SVS16` refers to a signed load and `SVU16` refers to an unsigned load. Note that the priming and flush instructions for unaligned stores and the special store instructions do not distinguish between signed and unsigned operations.

Table 45. Vectra LX Aligned Vector Store Instructions

Instruction	Operands	Description
SV{S,U}16.I	vr, ars, imm8	Store Vector (base + immediate)
SV{S,U}32.I	vr, ars, imm8	
SV{S,U}16.IU	vr, ars, imm8	Store Vector (base + immediate with update)
SV{S,U}32.IU	vr, ars, imm8	
SV{S,U}16.X	vr, ars, art	Store Vector (base + index)
SV{S,U}32.X	vr, ars, art	
SV{S,U}16.XU	vr, ars, art	Store Vector (base + index with update)
SV{S,U}32.XU	vr, ars, art	
SV{S,U}16.CU	vr, ars	Store Vector Circular, unit stride with update
SV{S,U}32.CU	vr, ars	

Table 46. Vectra LX Unaligned Vector Store Instructions

Instruction	Operands	Description
SVA.F	uu, ars	Store Vector Align Flush
SV{S,U}16A.I	vr, uu, ars, imm4	Store Vector Align (base + immediate)
SV{S,U}32A.I	vr, uu, ars, imm4	
SV{S,U}16A.IU	vr, uu, ars, imm4	Store Vector Align (base + immediate with update)
SV{S,U}32A.IU	vr, uu, ars, imm4	
SV{S,U}16A.X	vr, uu, ars, art	Store Vector Align (base + index)
SV{S,U}32A.X	vr, uu, ars, art	
SV{S,U}16A.XU	vr, uu, ars, art	Store Vector (base + index with update)
SV{S,U}32A.XU	vr, uu, ars, art	
SV{S,U}16A.CU	vr, uu, ars	Store Vector Align Circular, unit stride with update
SV{S,U}32A.CU	vr, uu, ars	

Table 47. Vectra LX Scalar Store Instructions

Instruction	Operands	Description
SS{S,U}16.I	vr, ars, imm8	Store Scalar (base + immediate)
SS{S,U}32.I	vr, ars, imm8	
SS{S,U}16.IU	vr, ars, imm8	Store Scalar (base + immediate with update)
SS{S,U}32.IU	vr, ars, imm8	
SS{S,U}16.X	vr, ars, art	Store Scalar (base + index)
SS{S,U}32.X	vr, ars, art	
SS{S,U}16.XU	vr, ars, art	Store Scalar (base + index with update)
SS{S,U}32.XU	vr, ars, art	

Table 48. Vectra LX Special Store Instructions

Instruction	Operands	Description
SVH.I	vr, ars, imm10	Store Vector High (base + immediate)
SVL.I	vr, ars, imm8	Store Vector Low (base + immediate)
SALIGN.I	uu, ars, imm8	Store Align Register (base + immediate)
SSEL.I	se, ars, imm8	Store Select Register (base + immediate)

2.16.10 Vectra LX ALU and Shift Instructions

Table 49. Vectra LX Arithmetic Instructions

Instruction	Operands	Description
ADD20	vt, vs, vr	Addition
ADD40	vt, vs, vr	Addition
SUB20	vt, vs, vr	Subtraction
SUB40	vt, vs, vr	Subtraction
MIN20	vt, vs, vr	Minimum (no Boolean register update)
MIN40	vt, vs, vr	Minimum (no Boolean register update)
MAX20	vt, vs, vr	Maximum (no Boolean register update)
MAX40	vt, vs, vr	Maximum (no Boolean register update)
ABS20	vt, vr	Absolute value
ABS40	vt, vr	Absolute value
NEG20	vt, vr	Negate
NEG40	vt, vr	Negate

Table 50. Vectra LX Logical Instructions

Instruction	Operands	Description
AND160	vt, vs, vr	Logical AND
NAND160	vt, vs, vr	Logical NAND
OR160	vt, vs, vr	Logical OR
XOR160	vt, vs, vr	Logical Exclusive OR

Table 51. Vectra LX Compare Instructions (with Boolean register update)

Instruction	Operands	Description
LT20	bb8, vs, vr	Less than
LT40	bb8, vs, vr	Less than
LE20	bb8, vs, vr	Less than or Equal
LE40	bb8, vs, vr	Less than or Equal
EQ20	bb8, vs, vr	Equal to
EQ40	bb8, vs, vr	Equal to

Table 52. Vectra LX Special ALU Instructions

Instruction	Operands	Description
MINB20	vt, vs, vr, ba8	Minimum with Boolean register update
MINB40	vt, vs, vr, ba4	Minimum with Boolean register update
MAXB20	vt, vs, vr, ba8	Maximum with Boolean register update
MAXB40	vt, vs, vr, ba8	Maximum with Boolean register update
RADD20	vt, vr	Reduction Add
RADD40	vt, vr	Reduction Add
MULSGN20	vt, vs, vr	Multiply Sign
MULSGN40	vt, vs, vr	Multiply Sign

Table 53. Vectra LX Shift Instructions

Instruction	Operands	Description
SLLI20	vt, vr, imm5	Shift Left Logical Immediate
SLLI40	vt, vs, imm5	Shift Left Logical Immediate
SLL20	vt, vr	Shift Left Logical (using VSAR)
SLL40	vt, vr	Shift Left Logical (using VSAR)
SRAI20	vt, vr, imm5	Shift Right Arithmetic Immediate
SRAI40	vt, vr	Shift Right Arithmetic Immediate
SRA20	vt, vr	Shift Right Arithmetic (using VSAR)
SRA40	vt, vr	Shift Right Arithmetic (using VSAR)
SLS20	vt, vr	Shift Left Saturate (using VSAR)
SLS40	vt, vr	Shift Left Saturate (using VSAR)

2.16.11 Vectra LX Multiply/Accumulate Instructions

Table 54. Vectra LX Real-Number Vector Multiplication Instructions

Instruction	Operands	Description
MUL18.0	vt, vs, vr	Multiply, no accumulate, even
MUL18.1	vt, vs, vr	Multiply, no accumulate, odd
MULA18.0	vt, vs, vr	Multiply and Accumulate, even
MULA18.1	vt, vs, vr	Multiply and Accumulate, odd
MULS18.0	vt, vs, vr	Multiply and Subtract, even
MULS18.1	vt, vs, vr	Multiply and Subtract, odd
MULR18.0	vt, vs, vr	Multiply and Round, even
MULR18.1	vt, vs, vr	Multiply and Round, odd

Table 55. Vectra LX Complex-Number Multiplication Instructions

Instruction	Operands	Description
RMUL18	vt, vs, vr	Real part of complex multiply, no accumulate
IMUL18	vt, vs, vr	Imaginary part of complex multiply, no accumulate
RMULA18	vt, vs, vr	Real part of complex multiply with accumulate
IMULA18	vt, vs, vr	Imaginary part of complex multiply with accumulate
RMULS18	vt, vs, vr	Real part of complex multiply with subtraction
IMULS18	vt, vs, vr	Imaginary part of complex multiply with subtraction
RMULR18	vt, vs, vr	Real part of complex multiply with rounding
IMULR18	vt, vs, vr	Complex part of multiply with rounding

Table 56. Vectra LX Move Instructions

Instruction	Operands	Description
MOVF20	vt, vs, br8	Move if (Boolean register) = false
MOVF40	vt, vs, br4	Move if (Boolean register) = false
MOVT20	vt, vs, br8	Move if (Boolean register) = true
MOVT40	vt, vs, br4	Move if (Boolean register) = true
MOV160	vt, vr	Move (unconditional)
MOVAR16	art, vr	Move element 0 of vector register to AR
MOVAR32	art, vr	Move element 0 of vector register to AR
MOVVR20	vr, art	Move AR to vector register
MOVVR40	vr, art	Move AR to vector register
MOVAB4	bt4, ars	Move 4 bits of AR to BR
MOVAB8	bt8, ars	Move 8 bits of AR to BR
MOVBA4	art, bs4	Move 4 bits of BR to AR
MOVBA8	art, bs8	Move 8 bits of BR to AR
MALIGN	uur, uut	Move alignment register to alignment register
MSEL	ser, set	Move select register to select register

Table 57. Vectra LX Miscellaneous Instructions

Instruction	Operands	Description
PACK40	vt, vs, vr	Pack signed 40-bit value into 20 bits
SEL	vt, vs, vr, se	Select
SELI	vt, vs, vr, imm3	Select with immediate
ESHFT20	vt, vs, vr	Shift vector register by one 20-bit element
ESHFT40	vt, vs, vr	Shift vector register by one 40-bit element
REP20	vt, vr, repsel	Replicate 20-bit scalar element 8 times
REP40	vt, vr, repsel	Replicate 40-bit scalar element 4 times
ZALIGN	uur	Zero out alignment register

Table 58. Vectra LX State Read and Write Instructions

Instruction	Description
RUR.CBEGIN	Read contents of CBEGIN state and save them to an AR register.
RUR.CEND	Read contents of CEND state and save them to an AR register.
RUR.ROUND_HI	Read bits [39:32] of the ROUND state and save them to an AR register.
RUR.ROUND_LO	Read bits [31:0] of the ROUND state and save them to an AR register.
RUR.VSAR	Read contents of VSAR state and save them to an AR register.
WUR.CBEGIN	Write contents of an AR register to the CBEGIN state.
WUR.CEND	Write contents of an AR register to the CEND state.
WUR.ROUND_HI	Write contents of an AR register to bits [39:32] of the ROUND state
WUR.ROUND_LO	Write contents of an AR register to bits [31:0] of the ROUND state
WUR.VSAR	Write contents of an AR register to the VSAR state.

2.17 Xtensa Application Binary Interface (ABI)

The Application Binary Interface (ABI) for Xtensa processors specifies the software conventions required for Xtensa code to interoperate. This section describes some highlights of the Xtensa ABI.

2.17.1 Data Types and Alignment

Table 59 shows the size and alignment for the standard C and C++ data types. Note that the `char` type is unsigned by default.

Table 59. Data Types and Alignment

Data Type	Size and Alignment
`char`	1 byte
`short`	2 bytes
`int`	4 bytes
`long`	4 bytes
`long long`	8 bytes
`float`	4 bytes
`double`	8 bytes
`long double`	8 bytes
`pointer`	4 bytes

2.17.2 Stack Frame Layout

The stack grows down from high to low addresses. The stack pointer must always be aligned to a 16-byte boundary. Every stack frame contains at least 16 bytes of register window save area. If a function call has arguments that do not fit in registers, they are passed on the stack beginning at the stack-pointer address. Figure 15 illustrates this stack-frame layout.

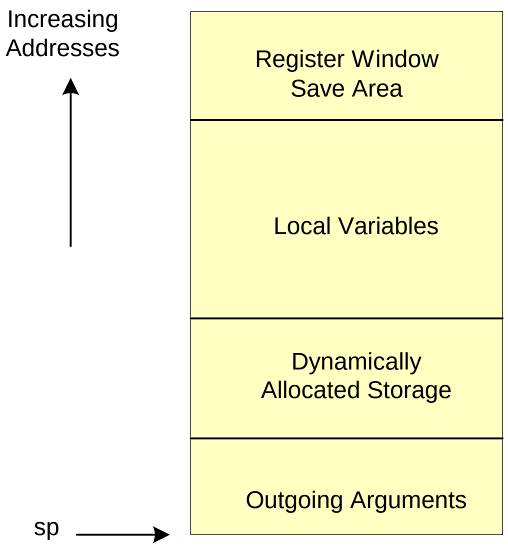


Figure 15. Stack Frame Layout

The frame-pointer register is only required when storage is dynamically allocated on the stack, for example, with the `alloca` instruction or for variable-sized local arrays. When a frame pointer is used, it is set to the value of the stack pointer before any stack space is dynamically allocated.

2.17.3 Calling Convention

The Xtensa windowed register calling convention is designed to efficiently pass arguments and return values in AR registers (see Table 60). The register windows for the caller and the callee are not the same, but they partially overlap. As many as six words of arguments can be passed from the caller to the callee in these overlapping registers, and as many as four words of a return value can be returned in the same registers. If all the arguments do not fit in registers, the rest are passed on the stack. Similarly, if the return value needs more than four words, the value is returned on the stack instead of the AR registers.

Table 60. General-Purpose Register Use

Register	Use
a0	Return Address
a1 (sp)	Stack Pointer
a2 – a7	Incoming Arguments
a7	Optional Frame Pointer

Example 1

```
int proc1 (int x, short y, char z) {
     // a2 = x
     // a3 = y
     // a4 = z
     // return value goes in a2
}
```

The registers that the caller uses for arguments and return values are determined by the size of the register window. The window size must be added to the register numbers seen by the callee. For example, if the caller uses a `CALL8` instruction, the window size is 8 and "`x = proc1 (1, 2, 3)`" translates to:

```
movi.n    a10, 1
movi.n    a11, 2
movi.n    a12, 3
call8     proc1
s32i      a10, sp, x_offset
```

Double-precision floating-point values and `long long` integers require two registers for storage. The calling convention requires that these registers be even/odd register pairs.

Example 2

```
double proc2 (int x, long long y) {
     // a2 = x
     // a3 = unused
     // a4,a5 = y
     // return value goes in a2/a3
}
```

3. Exception-Processing States and Options

The Xtensa LX processor implements basic functions needed to manage both *synchronous exceptions* and *asynchronous exceptions* (interrupts). However, the baseline processor configuration only supports synchronous exceptions, for example, those caused by illegal instructions, system calls, instruction-fetch errors, and load/store errors. Asynchronous exceptions are supported by processor configurations built with the interrupt, high-priority interrupt, and timer options.

The Xtensa LX interrupt option implements Level-1 interrupts, which are asynchronous exceptions triggered by processor input signals or software exceptions. Level-1 interrupts have the lowest priority of all interrupts and are handled differently than high-priority interrupts, which occur at priority levels 2 through 15. The interrupt option is a prerequisite for the high-priority-interrupt, peripheral-timer, and debug options.

The high-priority-interrupt option implements a configurable number of interrupt levels (2 through 15). In addition, an optional non-maskable interrupt (NMI) has an implicit infinite priority level. Like Level-1 interrupts, high-priority interrupts can be external or software exceptions. Unlike Level-1 interrupts, each high-priority interrupt level has its own interrupt vector and dedicated registers for saving processor state. These separate interrupt vectors and special registers permit the creation of very efficient handler mechanisms that may need only a few lines of assembler code, avoiding the need for a subroutine call.

3.1 Exception Architecture

The Xtensa LX processor supports one exception model, Xtensa Exception Architecture 2 (XEA2).

3.1.1 XEA2 Exceptions

XEA2 exceptions share the use of two exception vectors. These two vector addresses, `UserExceptionVector` and `KernelExceptionVector` are both configuration options. The exception-handling process saves the address of the instruction causing the exception into special register `EPC[1]` and the cause code for the exception into special register `EXCCAUSE`. Interrupts and exceptions are precise, so that on returning from the exception handler, program execution can continue exactly where it left off. (Note: Of necessity, write bus-error exceptions are not precise.)

3.1.2 Unaligned Exception Option

This option allows an exception to be raised upon any unaligned memory access whether it is generated by core processor memory instructions, by optional instructions, or by designer-defined TIE instructions. With the cooperation of system software, occasional unaligned accesses can be handled correctly.

Note that instructions that deal with the cache lines such as prefetch and cache-management instructions will not cause unaligned exceptions.

3.1.3 Interrupt Option

The Xtensa LX exception option implements Level-1 interrupts. The processor configuration process allows for a variable number of interrupts to be defined. External interrupt inputs can be level-sensitive or edge-triggered. The processor can test the state of these interrupt inputs at any time by reading the `INTERRUPT` register. An arbitrary number of software interrupts (up to a total of 32 for all types of interrupts) that are not tied to an external signal can also be configured. These interrupts use the general exception/interrupt handler. Software can then manipulate the bits of interrupt enable fields to prioritize the exceptions. The processor accepts an interrupt when an `interrupt` signal is asserted and the proper interrupt enable bits are set in the `INTENABLE` register and in the `INTLEVEL` field of the `PS` register.

3.1.4 High-Priority Interrupt Option

A configured Xtensa LX processor can have as many as 6 level-sensitive or edge-triggered high-priority interrupts. One NMI or non-maskable interrupt can also be configured. The processor can have as many as six high-priority interrupt levels and each high-priority interrupt level has its own interrupt vector. Each high-priority interrupt level also has its own dedicated set of three special registers (`EPC`, `EPS`, and `EXCSAVE`) used to save the processor state.

3.1.5 Timer Interrupt Option

The Timer Interrupt Option creates one to three hardware timers that can be used to generate periodic interrupts. The timers can be configured to generate either level-one or high-level interrupts. Software manages the timers through one special register that contains the processor core’s current clock count (which increments each clock cycle) and additional special registers (one for each of the three optional timers) for setting match-count values that determine when the next timer interrupt will occur.

4. Memory-Management Model Types and Options

The Xtensa LX processor core supports one memory-management configuration: region protection. This function is implemented by a block called the Region Protection Unit (RPU).

4.1 Region Protection

As shown in Table 61, the Region Protection Unit (RPU) divides the processor's memory space into eight memory regions. The RPU maintains separate protection settings for instruction and data memory accesses. The access modes of each memory region can be changed by issuing the appropriate `WITLB` or `WDTLB` instruction.

Table 61. Memory Map for Region Protection Option

Protected Region Number	Protected Region Starting Address
0	0x0000 0000
1	0x2000 0000
2	0x4000 0000
3	0x6000 0000
4	0x8000 0000
5	0xa000 0000
6	0xc000 0000
7	0xe000 0000

4.2 Access Modes

Table 62 lists and describes the supported access mode settings for instruction and data regions.

Table 62. Access Modes with Region Protection

Access Mode Value	Name	Instruction Fetch Behavior	Load Behavior	Store Behavior
0000	No allocate	Exception (`InstFetch Prohibited Cause`)	No allocate	Write-through/No write allocate
0001	Write-through/No write allocate	Allocate	Allocate	Write-through/No write allocate
0010	Bypass	Bypass	Bypass	Bypass
0011*	Mode not supported	Undefined	Exception (`LoadProhibitedCause`)	Exception (`StoreProhibitedCause`)
0100	Write-back mapped region with Write-back cache option	Allocate	Allocate	Write-back/Write allocate
	Write-back mapped region without Write-back cache option	Allocate	Allocate	Write-through/No write allocate
0101 - 1101	Reserved	Exception (`InstFetch Prohibited Cause`)	Exception (`LoadProhibitedCause`)	Exception (`StoreProhibitedCause`)
1110	Isolate	Exception (`InstFetch Prohibited Cause`)	Direct processor access to the memory cache	Direct processor access to the memory cache
1111	Illegal	Exception (`InstFetch Prohibited Cause`)	Exception (`LoadProhibitedCause`)	Exception (`StoreProhibitedCause`)

Notes:

RESET VALUE: 0010 (Bypass) for all regions.

* Access Mode 3 is not supported in the Xtensa LX core.

When an access mode attribute is set to a reserved or illegal value, memory accesses to that region cause an exception, as listed in the table above.

4.3 Protections

Each address and data memory segment is protected through the access modes (listed in Table 63) written into the RPU.

Table 63. Cache Access Mode Descriptions

Access Mode	Description
Uncached	Used for devices such as memory-mapped I/O devices whose memory addresses should never be cached.
No allocate	Do not allocate a line in the cache for this address. However, if the target address is already cached, use the cached value. If there is a cache line allocated for this address already but the value is not already in the cache, fetch the value from memory and load it into the cache.
Bypass	Do not use the cache.
Write-back	Write the value to the cache. The cached value is written to the appropriate memory address only when the corresponding cache line is evicted from the cache or when the processor forces the line to be written from the cache.
Write-through	Write the value to the cache and to the target memory address simultaneously.
Isolate	This mode permits direct load/store access to the cache memory and is used for manufacturing-test purposes.
Illegal	Access to an address marked as illegal causes an exception. This access mode can be used as a mechanism for protecting memory blocks.

5. Multiple Processor (MP) Features and Options

A multiple-processor (MP) option for the Xtensa LX architecture adds a processor ID register, break-in/break-out capabilities to the processor hardware, and optional instructions for MP synchronization. The Xtensa LX MP option also adds MP capability to the Xtensa LX instruction-set simulator (ISS) through the addition of the XTMP (Xtensa Modeling Protocol) simulation APIs. All versions of the Xtensa LX processor have certain optional PIF operations that enhance support for MP systems.

5.1 Processor ID Register

Some SOC designs use multiple Xtensa LX processors that execute from the same instruction space. The processor ID option helps software distinguish one processor from another via a `PRID` special register.

5.2 Multiple Processor On-Chip Debug with Break-in/Break-out

Placing multiple processors on the same IC die introduces significant complexity in SOC software debugging. A simplified method of debugging multiple-processor designs is available for those customers who have purchased the System Modeling and Multiple-Processor Support option. This capability enables one Xtensa processor to selectively communicate a Break to other Xtensa processors in a multiple-processor system. Two signals and one register have been introduced to achieve this ability.

The `XOCDModePulse` signal is a one-cycle OCD output pulse (Break-out) used to communicate a break command to other processors when the processor has entered `OCDHaltMode`. The `TDebugInterrupt` signal (Break-in), when asserted, will cause the processor core to take a debug interrupt if the break-in feature is enabled.

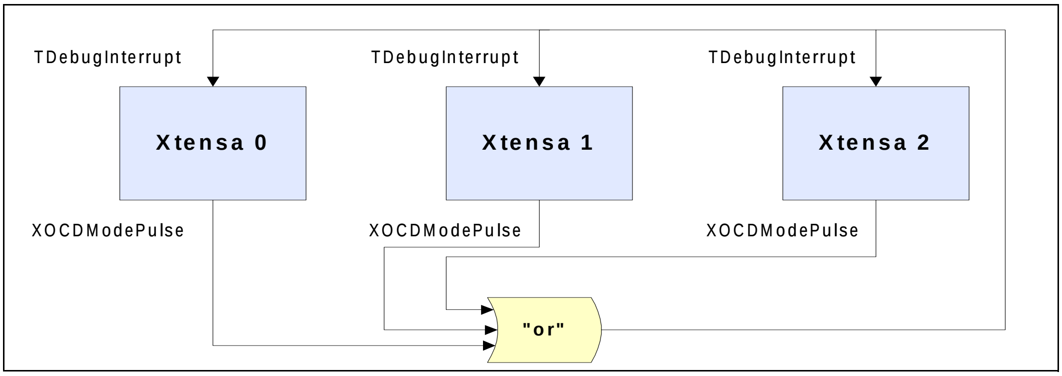


Figure 16. Designer-Defined On-Chip Debug Topology for Selective Break

For example, in Figure 16, the system designer adds logic (in this example, this could be as simple as an “OR” gate), the `XOCDModePulse` signal (break-out), and the `TDebugInterrupt` signal (break-in) to orchestrate the break sequence during on-chip debugging of a 3-processor core design.

The Debug Control Register (DCR) controls the break-in and break-out signals. The DCR contains two fields that control the break-in/break-out function. The `XOCDModePulseEn` bit controls whether the current processor will assert the `XOCDModePulse` signal when a break occurs in `OCDHaltMode`. The `extDbgIntEn` bit determines whether the current processor will enter `OCDHaltMode` when the `TDebugInterupt` signal is received. The behavior of these OCD signals depends on whether OCD has been enabled. For example, if OCD is disabled, the `XOCDModePulse` signal will not be asserted and the `TDebugInterupt` signal will be ignored. SOC designers will generally develop special logic that works with the Xtensa LX break-in/break-out signals to achieve the desired multiple-processor break sequence.

5.3 Multiprocessor Synchronization Option

When multiple processors are used in a system, some sort of communication and synchronization between processors is required. (Note that multiprocessor synchronization is distinct from pipeline synchronization between instructions as represented by the `ISYNC`, `RSYNC`, `ESYNC`, and `DSYNC` instructions, despite the name similarity.) In some cases, self-synchronizing communication structures such as input and output queues are used. In other cases, a shared-memory model is used for communication. Shared-memory communication protocols require instruction-set support for synchronization because shared memory does not provide the required semantics. The Xtensa LX Multiprocessor Synchronization Option provides instruction-set support for shared-memory communication protocols.

5.3.1 Memory-Access Ordering

The Xtensa ISA requires that valid programs follow a simplified version of the Release Consistency model of memory-access ordering. The Xtensa version of Release Consistency is adapted from *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors* by Gharachorlo et. al. in the Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990, from which the following three definitions are directly borrowed:

- A load by processor *i* is considered *performed with respect to processor k* at a point in time when the issuing of a store to the same address by processor k cannot affect the value returned by the load.
- A store by processor i is considered *performed with respect to processor k* at a point in time when an issued load to the same address by processor k returns the value defined by this store (or a subsequent store to the same location).
- An access is *performed* when it is performed with respect to all processors.

Using these definitions, the Xtensa LX processor places the following requirements on memory access:

- Before an ordinary load or store access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed, and
- Before a *release* access is allowed to perform with respect to any other processor, all previous ordinary load, store, acquire, and release accesses must be performed, and
- Before an acquire is allowed to perform with respect to any other processor, all previous acquire accesses must be performed.

Future Xtensa processor implementations may adopt stricter memory orderings for simplicity. Software programs, however, should not rely on any stricter memory ordering semantics than those specified here.

5.3.2 Architectural Additions

Table 64 shows the Multiprocessor Synchronization option's architectural additions.

Table 64. Multiprocessor Synchronization Option Instruction Additions

Instruction	Format	Definition
`L32AI`	RRI8	Load 32-bit Acquire (8-bit shifted offset) This non-speculative load will perform before any subsequent loads, stores, or acquires are performed. It is typically used to test the synchronization variable protecting a mutual-exclusion region (for example, to acquire a lock).
`S32RI`	RRI8	Store 32-bit Release (8-bit shifted offset) All prior loads, stores, acquires, and releases will be performed before this store is performed. It is typically used to write a synchronization variable to indicate that this processor is no longer in a mutual-exclusion region (for example, to release a lock).

5.3.3 Inter-Processor Communication with the `L32AI` and `S32RI` Instructions

The 32-bit load and store instructions `L32AI` and `S32RI` add acquire and release semantics to the Xtensa LX processor. These instructions are useful for controlling the ordering of memory references in multiprocessor systems, where different memory locations can be used for synchronization and data and precise ordering between synchronization references must be maintained. Other Xtensa LX load and store instructions may be executed by processor implementations in any order that produces the same uniprocessor result.

`L32AI` is used to load a synchronization variable. This load will be performed before any subsequent load, store, acquire, or release is begun. This characteristic ensures that subsequent loads and stores do not see or modify data that is protected by the synchronization variable.

`S32RI` is used to store to a synchronization variable. This store will not begin until all previous loads, stores, acquires, or releases are performed. This characteristic ensures that any loads of the synchronization variable that see the new value will also find all protected data available as well.

Consider the following example:

```
volatile uint incount = 0;
volatile uint outcount = 0;
const uint bsize = 8;
data_t buffer[bsize];
void producer (uint n)
{
   for (uint i = 0; i < n; i += 1) {
      data_t d = newdata();            // produce next datum
      while (outcount == i - bsize);   // wait for room
      buffer[i % bsize] = d;           // put data in buffer
      incount = i+1;                   // signal data is ready
   }
}
void consumer (uint n)
{
   for (uint i = 0; i < n; i += 1) {
      while (incount == i);            // wait for data
      data_t d = buffer[i % bsize];    // read next datum
      outcount = i+1;                  // signal data read
      usedata (d);                     // use datum
   }
}
```

Here, `incount` and `outcount` are synchronization variables, and `buffer` is a shared data variable. `producer`'s writes to `incount` and `consumer`'s writes to `outcount` must use `S32RI` and `producer`'s reads of `outcount` and `consumer`'s reads of `incount` must use `L32AI`. If `producer`'s write to `incount` were performed with a simple `S32I`, the processor or memory system might reorder the write to `buffer` after the write to `incount`, thereby allowing `consumer` to see the wrong data. Similarly, if consumer's read of `incount` were done with a simple `L32I`, the processor or memory system might reorder the read to `buffer` before the read of `incount`, also causing `consumer` to see the wrong data.

5.4 Conditional Store Option

In addition to the memory ordering needs satisfied by the Multiprocessor Synchronization Option (page 79), a multiprocessor system can require mutual exclusion, which cannot easily be programmed using the Multiprocessor Synchronization Option. The Conditional Store Option is intended to add that capability. It does so by adding a single instruction (`S32C1I`) that atomically stores to a memory location only if its current value is the expected one. A state register (`SCOMPARE1`) is also added to provide the additional operand required.

Note: The Conditional Store option requires that the bus between the Xtensa LX processor and shared memory and the shared memory itself be able to handle atomic read-compare-write operations.

5.4.1 Architectural Additions

Table 65 and Table 66 show the Conditional Store option's architectural additions.

Table 65. Conditional Store Option Processor-State Additions[1]

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
SCOMPARE1	1	32	Expected Data Value for S32C1I Instruction	R/W	12

1. Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. .

Table 66. Conditional Store Option Instruction Additions

Instruction	Format	Definition
S32C1I	RRI8	Store 32-Bit Compare Conditional Stores to a location only if the location contains the value in the SCOMPARE1 register. The comparison of the old value and the store, if equal, is atomic. The instruction also returns the old value of the memory location.

5.4.2 Exclusive Access with the S32C1I Instruction

L32AI and S32RI allow inter-processor communication, as in the producer-consumer example in Section 5.3.3 on page 82 (barrier synchronization is another example), but they are not efficient for guaranteeing exclusive access to data (for example, locks). Some systems may provide efficient, tailored, application-specific exclusion support. When this is not appropriate, the ISA provides another general-purpose mechanism for atomic updates of memory-based synchronization variables that can be used for exclusion algorithms. The S32C1I instruction stores to a location if the location contains the value in the SCOMPARE1 register. The comparison of the old value and the store, if equal, are atomic. S32C1I also returns the old value of the memory location, so it looks like both a load and a store; this allows the program to determine whether the store succeeded, and if not it can use the new value as the comparison for the next S32C1I. For example, an atomic increment could be done as follows:

```
        l32ai     a3, a2, 0           // current value of memory
   loop:
        wsr       a3, scompare1       // put current value in SCOMPARE1
        mov       a4, a3              // save for comparison
        addi      a3, a3, 1           // increment value
        s32c1i    a3, a2, 0           // store new value if memory
                                      // still contains SCOMPARE1
        bne       a3, a4, loop        // if value changed, try again
```

Semaphores and other exclusion operations are equally simple to create using `S32C1I`.

`S32C1I` may be used only on cache bypass memory locations for which old and new data are sent to memory for the compare and possible write. It may not be used on cacheable memory locations, DataRAM/DataROM addresses, instruction addresses, or XLMI addresses.

There are many possible atomic memory primitives. `S32C1I` was chosen for the Xtensa ISA because it can easily synthesize all other primitives that operate on a single memory location. Many other primitives (for example, test and set, or fetch and add) are not as universal. Only primitives that operate on multiple memory locations are more powerful than `S32C1I`.

Note: The `SCOMPARE1` register is undefined after Reset.

5.4.3 Memory Ordering and the S32C1I Instruction

With regard to the memory ordering defined for `L32AI` and `S32RI` in Section 5.3.1 on page 81, `S32C1I` plays the role of both acquire and release. That is, before the atomic pair of memory accesses can perform, all ordinary loads, stores, acquires, and releases must have performed. In addition, before any following ordinary load, store, acquire, or release can be allowed to perform, the atomic pair of the `S32C1I` must have performed. This allows the Conditional Store to make atomic changes to variables with ordering requirements, such as the counts discussed in the example in Section 5.3.3 on page 82.

5.5 Inbound Processor Interface (PIF) Operations

An Xtensa LX core can optionally accept inbound requests from the PIF that access the local memories or the XLMI port. The processor must have a PIF for the inbound-PIF Request option to be enabled. The processor must also have at least one instruction RAM, data RAM, or an XLMI port to be compatible with this option. Inbound-PIF requests are configured separately for each configured local memory or XLMI port.

Using inbound-PIF operations, Xtensa LX processors can write into and read from each other’s local memories if the processors PIFs are connected to a common bus, a feature that can be used for relatively “glue-less” MP communications.

Inbound-PIF operations directed at local data memories use the Xtensa LX processor’s load/store unit to effect the requested transfers. Inbound-PIF operations directed at local instruction memories do not use the Xtensa LX processor’s load/store unit.

If the Xtensa LX configuration includes the optional second load/store unit, inbound-PIF operations will attempt to use that second unit for transfers.

5.5.1 Simultaneous Memory-Access and Inbound-PIF Operations

Core processor instructions and inbound-PIF transfers can both use the Xtensa LX processor’s first load/store unit simultaneously. This feature can be used to significantly boost system performance. The bandwidth allocated by the processor to inbound-PIF operations can be allocated under program control.

6. Xtensa Software Development Tools and Environment

Xtensa processors are supported by industry-leading software development tools including a complete C development tool chain, integrated development environment, and other debugging and development tooling. Using the Xtensa software development tools, you can compile, run and debug your C/C++ and assembly application on the single-processor Instruction Set Simulator (ISS), multiple-processor XTMP simulation environment, or real Xtensa hardware. You can model your own system using XTMP and you can view performance of your application. Xtensa processors have full support from leading Real Time Operating System vendors, including automatic support for each unique Xtensa processor configuration.

6.1 Xtensa Xplorer IDE

Xtensa Xplorer serves as a cockpit for multiple-processor SOC hardware and software design. Xtensa Xplorer integrates software development, processor optimization, and multiple-processor SOC architecture tools into one common design environment. It also integrates SOC simulation and analysis tools. Xtensa Xplorer is a visual environment with a host of automation tools that makes creating Xtensa processor-based SOC hardware and software much easier.

Xtensa Xplorer is particularly useful for the development of TIE instructions that maximize performance for a particular application. Different Xtensa processor and TIE configurations can be saved, profiled against the target C/C++ software, and compared. Xtensa Xplorer even includes automated graphing tools that create spreadsheet-style comparison charts of performance.

Xtensa Xplorer dramatically accelerates the processor optimization cycle by providing an automated, visual means of profiling and comparing different processor configurations. With Xtensa Xplorer, it takes only a few minutes to try out a new instruction and get valuable feedback, right at the designer’s desktop. Then, after all the instructions are evaluated, it takes only about an hour for the Xtensa Processor Generator to create a new, customized version of the Xtensa processor, including all of the custom instructions plus the associated software environment. This makes it easy for designers to try many different options.

Xtensa Xplorer helps bridge the hardware-software design gap for extended instructions by providing a context-sensitive TIE code editor, a TIE-instruction-aware debugger, and a gate-count estimator. The gate-count estimator gives real-time feedback to software designers unfamiliar with hardware development, allowing them to explore various embedded processor instruction set architectures while receiving immediate feedback on the cost implications of their extended-instruction choices.

Xtensa Xplorer, while extremely useful for designs with only one Xtensa processor, is ideally targeted to multiple-processor SOC (MPSOC) designs. It facilitates MPSOC development with tools for build management; profiling; batch building; system memory map assignment; and integrated multiple-processor simulation, creation, and debugging using Tensilica's Xtensa Modeling Protocol (XTMP).

Xtensa Xplorer also provides a unified environment for:

- C/C++ application software development
- C/C++ program debugging
- Code profiling and visualization
- SOC configuration management (using multiple Xtensa processors in a system)

Xtensa Xplorer automatically generates and manages makefiles for single and multiple processor systems of configured processors. It provides a graphical debugger and extensive performance visualization tools. It also has system-construction tools that generate XTMP simulator source code from simple, tabular system descriptions.

Xtensa Xplorer can automatically drive designer-directed exploration of different processor configurations including analysis of application performance under different cache configurations as well as analysis of different instruction sets generated by the XPRES compiler. Xplorer, which is based on the Eclipse universal tool platform, has plug-ins available from third parties that support automatic integration with most source-code management systems.

The configuration capabilities of previous releases of the Xtensa Processor Generator (a Web-based application) are now integrated into the Xtensa Xplorer (an application that runs locally on the designer's PC or workstation). Xtensa Xplorer integrates all of the processor-configuration steps (configure, build, download, install, and iterate) into one seamless design environment.

6.2 Xtensa C and C++ Compiler (XCC)

The Xtensa C and C++ Compiler (XCC) is an optimized compiler for the Tensilica Xtensa processor. Operation of XCC is similar to operation of standard GNU C and C++ compilers (GCC). XCC provides superior code execution performance and reduced code size compared to GCC through improved optimization and code generation technology. XCC delivers enormous benefits to users of Tensilica's XPRES Compiler and users of the Vectra LX DSP Engine through automated vectorization of C and C++ source code.

6.2.1 Profiling Feedback

The Xtensa Tools Version 6 release of XCC adds the ability to use run-time statistics to improve the results delivered by XCC. Many tradeoffs made by a compiler in optimizing code are affected by the direction of branches and the relative execution frequency of different source functions. Profiling information allows XCC to make much better estimates of code behavior.

To use this feature, you first compile your application using a compiler flag that directs the compiler to instrument your application so that you can generate profiling statistics. Then you run your application using a representative data set or data sets to generate the statistics. The closer the data set or sets match the data expected by your actual application, the better the compiler will do. Your application can be run in the ISS or any hardware system that supports a basic file system. After generating the feedback files with the profiling statistics, you recompile your application using another compiler flag that optimizes the code using the profiling information. The other compiler-optimization flags and the source code should not be changed without regenerating the feedback files. If you attempt to use obsolete feedback files, the compiler will warn you.

Feedback can significantly improve run-time performance, but can have an even greater effect on code size because it allows the compiler to decide which regions of code to compile for speed and which regions to compile for compactness.

6.2.2 Interprocedural Analysis (IPA)

Interprocedural analysis (IPA) allows the compiler to evaluate multiple C or C++ source files together as a group during optimization and thus allows the compiler to improve performance when compiling for speed and to reduce code size when compiling for size. To invoke IPA, you simply use the `-ipa` flag during both the compile and link steps. Individual source files can still be compiled separately. Behind the scenes, IPA delays optimization until the link step.

6.3 GNU Software Development Tools

Many of the Xtensa version 6 software development tools are built around the Free Software Foundation's GNU tools. The following components are included in the Xtensa LX software-development tool suite:

- Assembler
- Linker
- Symbolic debugger

- Binary utilities
- Profiler
- Standard libraries

6.3.1 Assembler

The GNU assembler translates Xtensa assembly source files into machine code using a standard ELF object file format. Besides the standard features of the GNU assembler, including a rich set of directives and a built-in macro processor, there are a number of special assembler features for Xtensa processors. For example, the assembler can:

- automatically align branch targets to improve performance
- translate pseudo-instructions into configuration-specific opcodes
- automatically convert instructions with out-of-range immediate operands into equivalent sequences of valid instructions
- use its scheduler to rearrange code and to bundle FLIX operations

Tensilica's version of the GNU assembler also includes a dynamic plug-in capability to make it easy to add designer-defined TIE instructions or switch Xtensa processor configurations.

6.3.2 Linker

The GNU linker combines object files and libraries into executable programs. Its behavior is controlled by linker scripts that give you flexible control over where the code and data are placed into memory. The Xtensa version of the GNU linker can also remove redundant literal values and optimize calls across object files.

Tensilica has introduced the concept of linker support packages (LSPs) as a way of organizing the linker scripts and runtime libraries associated with different target environments. During development, embedded software is often run using a variety of environments—functional testing with simulation, hardware verification, emulation boards, etc.—and these environments may require different memory layouts and runtime libraries. LSPs provide an easy way to organize support for these different runtime environments.

Besides providing several default LSPs, Tensilica supplies a tool to simplify the process of creating new linker scripts. This tool allows you to specify a system memory map in a simple format without needing to learn how to write linker scripts directly. Of course, if you need the full power and control of custom linker scripts, you can also create new LSPs with your own linker scripts.

6.3.3 Debugger

The GNU debugger (GDB) provides sophisticated symbolic debugging for C, C++, and assembly-language programs. You can access GDB directly with a command-line interface, or you can control GDB via the Xtensa Xplorer integrated development environment (IDE). GDB provides many features to control the execution of a program and to analyze its behavior. These features include:

- Breakpoints (with or without conditions)
- Watchpoints (to stop whenever the value of an expression changes)
- Single-stepping (by either source lines or machine instructions)
- Stack backtracing
- Examining memory (in terms of either raw values or high-level data structures)
- Viewing processor registers and other processor state
- Integrated disassembler

Tensilica's version of GDB supports debugging on various target systems, including the Xtensa Instruction Set Simulator (ISS), emulation boards (e.g, XT2000) via either a serial cable or a network connection, and Xtensa on-chip debug (OCD) targets.

6.3.4 Binary Utilities

The GNU binary utilities are a set of various tools that manipulate and analyze binary object files. These tools are an essential part of the software development environment. They include tools for:

- Building and modifying archive libraries of object files
- Transforming object files in various ways, such as stripping debugging information or translating to a different binary format (for example, S-records)
- Extracting information from object files, including symbol tables, size information, ELF headers, and disassembled code

6.3.5 Profiler

Profiling allows you to tune the performance of a program by locating the portions that consume the most execution time, so that you can focus your optimization effort where it matters most. You can tune Xtensa LX code using the traditional approach to software tuning (by optimizing the C, C++, and assembly language code) or with TIE. Tuning with TIE is likely to produce much faster code.

The GNU profiler uses information collected during the program's execution and summarizes it for analysis. Several output forms are available. The flat profile output shows the number of invocations and time spent in every function. The call graph output shows the number of times a function is invoked from various call sites, the number of times it calls other functions, and estimates of the execution time attributable to each function and its descendants.

On other systems, the GNU profiler relies on statistical approximations, but Tensilica uses a customized approach that avoids these inaccuracies. Instead of the usual statistical sampling, Tensilica uses the Xtensa Instruction Set Simulator (ISS) to collect profile data as every instruction is executed.

Besides profiling execution time, the Xtensa ISS collects information about other events such as cache misses and pipeline interlocks. The GNU profiler can then analyze these events. This feature provides you with an additional level of detail to better understand a program's performance.

6.3.6 Standard Libraries

Tensilica's software-development environment includes a complete set of standard libraries for use with your application programs.

▪ Standard C++ Library	This is the same library used by the popular GNU C++ compiler.
▪ C Library (libc)	Red Hat's "newlib" C library, which is designed specifically for use in embedded systems.
▪ Math Library (libm)	A complete set of standard mathematical functions, also from Red Hat's "newlib" package.
▪ Compiler Support Library (libgcc)	Runtime support functions for use by the compiler. This library comes from the GNU compiler, but Tensilica has added hand-optimized, configuration-specific implementations of the integer multiply and divide functions. The floating-point emulation functions are from the "SoftFloat" package, which provides very efficient implementations of the standard IEEE floating-point operations.

Note that some third-party operating systems may provide their own implementations of the above libraries.

6.4 The xt-trace Tool

The Xtensa LX trace port is a hardware configuration option. Its signals reflect the processor's internal state during operation. The xt-trace software tool converts raw trace information from the trace port into a stream of assembly instructions. It also provides varying levels of information about the processor's state.

6.5 Instruction Set Simulator (ISS) and Xtensa Modeling Protocol (XTMP) Environment

Tensilica's advanced Instruction Set Simulator (ISS) and Xtensa Modeling Protocol (XTMP) environment enable fast and accurate simulation of system-on-chip designs incorporating one or more processor cores. Running up to 100x faster than RTL simulators, the ISS/XTMP environment is a powerful tool for software development and SOC design. The ISS is also flexible: it can serve as a back end for the debugger, generate profiling data for Tensilica's `gprof` profiler, or operate as a system-simulation component using the XTMP environment or third-party co-simulation products such as Mentor Graphics Seamless Co-Verification Environment.

The XTMP environment allows system designers to create a customized, multi-threaded simulator to model complicated systems. Designers can instantiate multiple Xtensa cores and use XTMP to connect them to designer-defined peripherals and interconnects. The Xtensa cores and peripherals can accurately communicate using a cycle-accurate, split-transaction simulation model. XTMP can be used for simulating homogeneous or heterogeneous multiple-processor systems as well as complex uni-processor system architectures. The ISS/XTMP environment allows system designers to create, debug, profile, and verify their combined SOC and software architecture early in the design process.

The standalone Xtensa ISS is a component of the Xtensa software tool suite and is used with the accompanying profiling tool for application tuning and processor configuration. The ISS is available in library form accessible either through the Xtensa co-simulation model or through the ISS/XTMP environment.

The Xtensa ISS is a software application that models the behavior of the Xtensa instruction set. While the ISS directly models the Xtensa pipeline and is thus quite accurate, unlike its counterpart processor hardware model (Verilog or VHDL code), the ISS abstracts the CPU during its instruction execution to speed simulation. As a result, there may be small differences in reported cycle counts between the ISS and actual hardware in special circumstances such as during inbound-PIF operations, which can disturb the normal pipeline operation. An ISS can run much faster than hardware simulation because it does not have to model every signal transition for every gate and register in the complete processor design.

The ISS executes programs generated for the Xtensa processor on a host computer. It accurately reproduces the processor's reset and interrupt behavior allowing low-level programs (for example, device drivers and initialization code) to be developed. This technique is particularly useful when porting native code to an embedded application, because the ISS can be used to unfold potential problems (architectural assumptions, memory ordering considerations, etc.) without having to download the actual code to real embedded target hardware.

Table 67 compares the speed of various simulation options you have when you design with the Xtensa LX processor, and shows the relatively high performance of the ISS/XTMP combination.

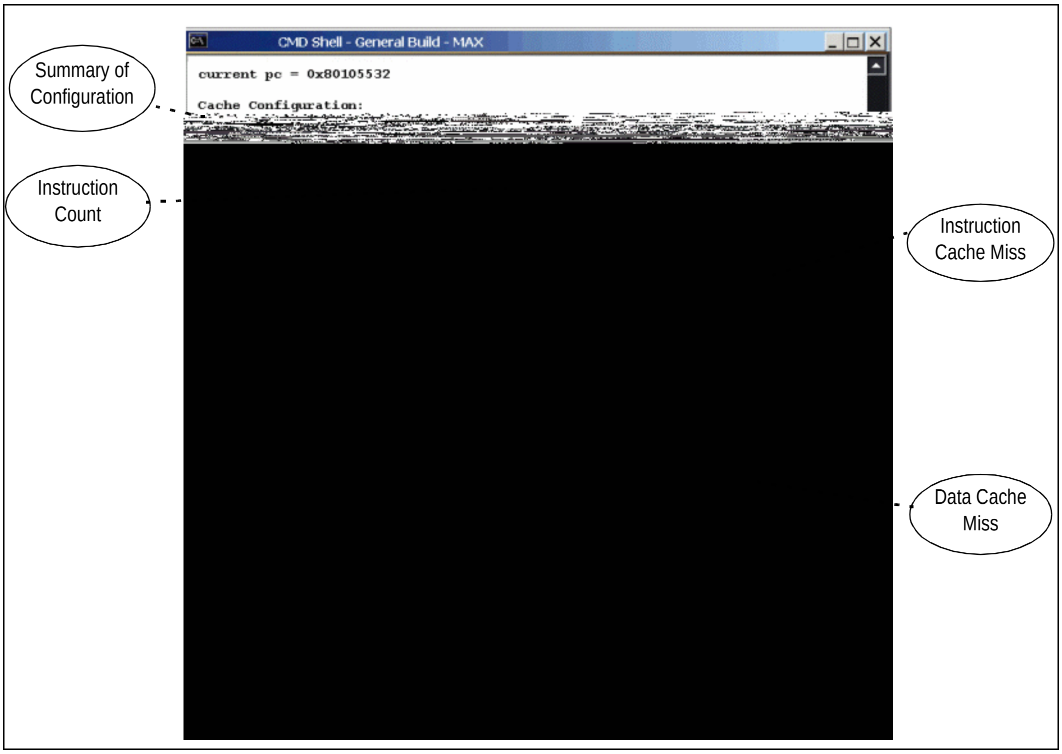
Table 67. Instruction Set Simulator Performance Comparisons with Other Tools

Simulation speed (cycles/seconds)	Modeling Tool	Benefits
200K to 400K*1, *2	ISS/XTMP	▪ Multiple processor subsystem modeling ▪ Pipeline-cycle accurate
200K to 400K*1	Standalone ISS	▪ Software verification ▪ Pipeline-cycle accurate ▪ High flexibility
2K to 10K *1	RTL Simulation	▪ Functional verification ▪ Pipeline-cycle accurate ▪ High visibility and accuracy
10 to 100 *1	Gate-level Simulation	▪ Timing verification ▪ Pipeline-cycle accurate ▪ High visibility and accuracy

*1 Simulation speed is an estimate, and is dependant on the computer. These estimates are based on a 2-GHz host PC running Linux.

*2 Simulation speed is an estimate for one ISS in the XTMP environment.

Figure 17 shows an example output from the instruction set simulator (ISS).



```
current pc = 0x80105532

Cache Configuration:
```

Figure 17. Instruction Set Simulator Sample Output

6.6 On-Chip Debug OCD Features

On-chip debug (OCD) is a useful method for developing and integrating application code and for SOC debug and validation. The Xtensa LX processor offers an optional OCD module that provides comprehensive debug support through a variety of features.

6.6.1 Functional Description of the OCD Module

The optional Xtensa LX on-chip debug module (OCD) provides external access to the internal, software-visible processor state through a narrow port connected directly to the processor core. The Xtensa LX OCD module uses the IEEE 1149.1 (JTAG) TAP port for communications with an external probe. Tensilica provides an example TAP controller to control the OCD module from a 5-pin JTAG TAP port. The OCD module incorporates several TAP registers that can be manipulated using OCD-specific TAP instructions. Using this 5-pin JTAG port, it is possible to do all of the following:

1. Generate an interrupt to put the processor in the debug mode
2. Read any program-visible register or memory location
3. Modify any program-visible register or memory location
4. Modify the processor's PC to jump to a desired code location
5. Return to normal operating mode

A host system (PC or Workstation) can be connected to the JTAG port through an intermediate board that converts host commands into JTAG commands and formats.

6.6.2 Uses of the OCD Module

OCD provides a powerful visibility tool into processor state. This tool can be used for various tasks such as debugging and profiling. The Xtensa software development environment supports the use of the OCD module with the GDB debugger with one of the Xtensa Microprocessor Emulation Kits.

6.6.3 Xtensa OCD Daemon

XOCD (the Xtensa OCD daemon) is a version of Macraigor's OCDemon® for Windows provided by Tensilica for the Xtensa processor architecture. It allows direct control over processor cores connected through the OCD (On-Chip Debug) port and does not require a stub running on the target. The target core can be stopped at any time to examine the state of the core, memory, and program. XOCD provides greater target visibility than monitor-based target debuggers and allows debugging of reset sequences, interrupts, and exceptions. XOCD implements a GDB-compatible stub interface and works with any debugger that supports the GDB remote protocol over a TCP/IP connection.

6.6.4 Multiple Processor On-Chip Debug with Break-in/Break-out

Placing multiple processors on the same IC die introduces significant complexity in SOC software debugging. A simplified method of debugging multiple-processor designs is available for those customers who have purchased the System Modeling and Multiple-Processor Support option. This capability enables one Xtensa processor to selectively communicate a Break to other Xtensa processors in a multiple-processor system. Two signals and one register have been introduced to achieve this ability.

The `XOCDModePulse` signal is a one-cycle OCD output pulse (Break-out) used to communicate a break command to other processors when the processor has entered `OCDHaltMode`. The `TDebugInterrupt` signal (Break-in), when asserted, will cause the processor core to take a debug interrupt if the break-in feature is enabled.

Figure 18. Designer-Defined On-Chip Debug Topology for Selective Break

For example, in Figure 18, the system designer adds logic (in this example, this could be as simple as an "OR" gate), the `XOCDModePulse` signal (break-out), and the `TDebugInterrupt` signal (break-in) to orchestrate the break sequence during on-chip debugging of a 3-processor core design.

The Debug Control Register (DCR) controls the break-in and break-out signals. The DCR contains two fields that control the break-in/break-out function. The `XOCDModePulseEn` bit controls whether the current processor will assert the `XOCDModePulse` signal when a break occurs in `OCDHaltMode`. The `extDbgIntEn` bit determines whether the current processor will enter `OCDHaltMode` when the `TDebugInterupt` signal is received. The behavior of these OCD signals depends on whether OCD has been enabled. For example, if OCD is disabled, the `XOCDModePulse` signal will not be asserted and the `TDebugInterupt` signal will be ignored. SOC designers will generally develop special logic that works with the Xtensa LX break-in/break-out signals to achieve the desired multiple-processor break sequence.

6.7 *The Xtensa Microprocessor Emulation Kit –XT2000-X Development Board*

The Xtensa LX processor is supplied with a full complement of tools for working with the emulation kit and XT2000-X board. These tools include the Xtensa monitor program, the Xtensa OCD daemon, board-configuration documentation, and an FPGA programming guide. The Xtensa Processor Generator also includes a build option that generates bitstream files for the FPGAs on the XT2000-X development board, providing automated support for reconfiguring the processor emulation on the XT2000-X board.

The XT2000 Emulation Kit uses complex FPGAs to emulate an Xtensa LX core processor configuration. This product allows a designer to evaluate various processor configuration options in a system environment and start software development and debug early in the design cycle. The FPGA on the XT2000 can easily accommodate the Vectra LX

DSP engine along with the Xtensa LX processor core. The XT2000 also offers additional expansion through PCI and Compact PCI connectors, as well as a large memory capacity through several on-board RAM and ROM sockets.

XT2000 Features:

- Programmable logic-based CPU Core emulation
- 15 to 33 MHz emulation frequency
- 256 Mbytes of SDRAM
- 3 PCI Slots
- Compact PCI slot
- 4 Mbytes of EPROM
- 1 Mbyte of SRAM
- 32 Mbytes of Flash memory, expandable to 64 Mbytes
- 2 Mbytes of Synchronous SRAM
- On-board Ethernet (10 base T)
- Two RS232 serial channels
- Battery backed-up real time clock
- Debug support
- Real-time trace function
- 8 character alpha-numeric LED
- Expansion connector
- Power supply

Tensilica's XT2000 Emulation Kit is a complete emulation, evaluation, and development kit for the Xtensa configurable core. The Tensilica Processor Generator is used to configure the core, which is then compiled with FPGA design tools to produce the FPGA configuration data. The FPGA configuration data is downloaded to the FPGA device residing on the emulation board. A predefined external bus interface connects the processor core through the processor core interface to the emulation board system resources.

The XT2000 also includes Red Hat's Redboot embedded bootstrap environment, which provides communications and debug facilities. An RS-232 serial port provides a communications link to a UNIX or PC host for downloading and debugging application programs under control of a host resident debugger.

The emulation kit is shipped with a default configuration that is useful for training and benchmarking. Figure 19 shows the XT2000 Emulation Kit's block diagram.

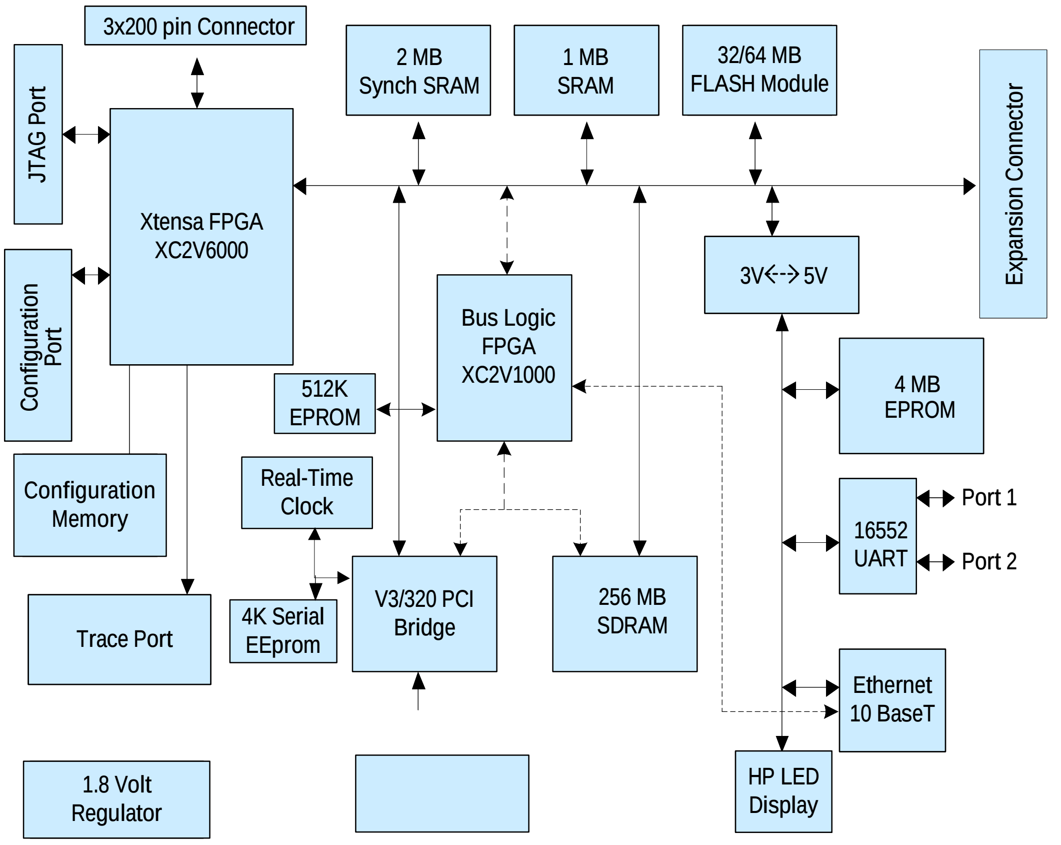


Figure 19. XT2000 Emulation Kit Block Diagram

6.8 RTOS Support and the OSKit™ Targeting Environment

Linker support packages and an OSKit RTOS targeting environment are available for Xtensa LX processors. The linker support package includes linker scripts, exception handler examples, and C runtime examples to allow the user to initiate fundamental kernel-development work. Xtensa LX RTOS support consists of a generic HAL (a hardware-abstraction layer consisting of configured header and library files), which is used by the Nucleus RTOS from Mentor Graphics/Accelerated Technology to support various configurations of the Xtensa LX processor, and a configured overlay for the VxWorks RTOS for Tornado from WindRiver Systems, which is included in the OSKit environment. This overlay includes hardware-abstraction layers and board-support packages for the ISS and the Xtensa Microprocessor Emulation Kit–XT2000.

7. TIE for Software Developers

This chapter introduces the Tensilica Instruction Extension (TIE) language and its features.

7.1 Adapting the Processor to the Task

Processors have traditionally been extremely difficult to design and modify. As a result, most systems contain processors that were designed and verified once for general-purpose use, and then embedded into multiple applications over time. It then becomes the software developer's responsibility to tune the application code to run well on this processor. Optimizing code to run efficiently on a general-purpose processor is a labor intensive process, with the developer spending countless hours trying out various techniques to make the code run faster. Would it not be better instead to have a processor whose instruction set can be customized for a specific application or class of applications?

The Tensilica Instruction Extension (TIE) language provides the software developer with a concise way of extending the Xtensa processor's architecture and instruction set. In most cases, the developer simply describes the semantics of a set of desired custom instructions. From this description, the Xtensa Processor Generator and the TIE compiler automatically generate the hardware that decodes the new instructions and integrates their execution semantics into the Xtensa pipeline. In addition, the Xtensa Processor Generator and the TIE compiler generate a complete set of software tools including a compiler, assembler, simulator, and debugger that are each customized for the generated processor core.

7.2 TIE Development Flow

Instruction-set extensions are typically used to accelerate both compute- and I/O-intensive functions, or "hot spots," in a particular piece of application code. Because a hot spot is a region of code that is exercised heavily by the program, the efficiency of its implementation can have a significant effect on the efficiency of the whole program. It is possible to increase the performance of an application by a factor of 2, 5, and sometimes by an order of magnitude or more by adding a few carefully selected instructions to the processor.

TIE can be used in two ways: through the Xtensa Processor Generator and locally, through the TIE compiler. Local use of the TIE compiler allows developers to rapidly experiment with new instructions to quickly tune and optimize application code using ISA extensions. Locally developed TIE then serves as an input to the Xtensa Processor Generator, which generates a processor with the ISA extensions and verifies the complete configuration.

Using the profiling tools available in the Xtensa Xplorer environment or using the `xt-gprof` utility together with the instruction set simulator (ISS), you can easily find an application's performance-critical code sections. Performance tuning is often an iterative process as illustrated in Figure 20. You can look at a region of code and experiment with different TIE instructions. After you have identified a critical code segment that can benefit from the addition of a TIE instruction, you describe the instruction in TIE and invoke the TIE compiler.

The TIE compiler generates a set of files that are used to configure all the software development tools so that they recognize your TIE extensions. The TIE compiler also provides a preliminary estimate of the hardware size of your new instruction to help you evaluate your new instruction. You now modify your application code as described in the next section to take advantage of the new instruction and simulate using the ISS to verify correct operation. You can look at the generated assembly code and rerun the profiler to evaluate performance. At this point you can decide if the performance improvement warrants the additional hardware cost and whether you should keep, remove, or modify the instruction. The turnaround time for this process is very short, because the software tools are automatically reconfigured for your new TIE.

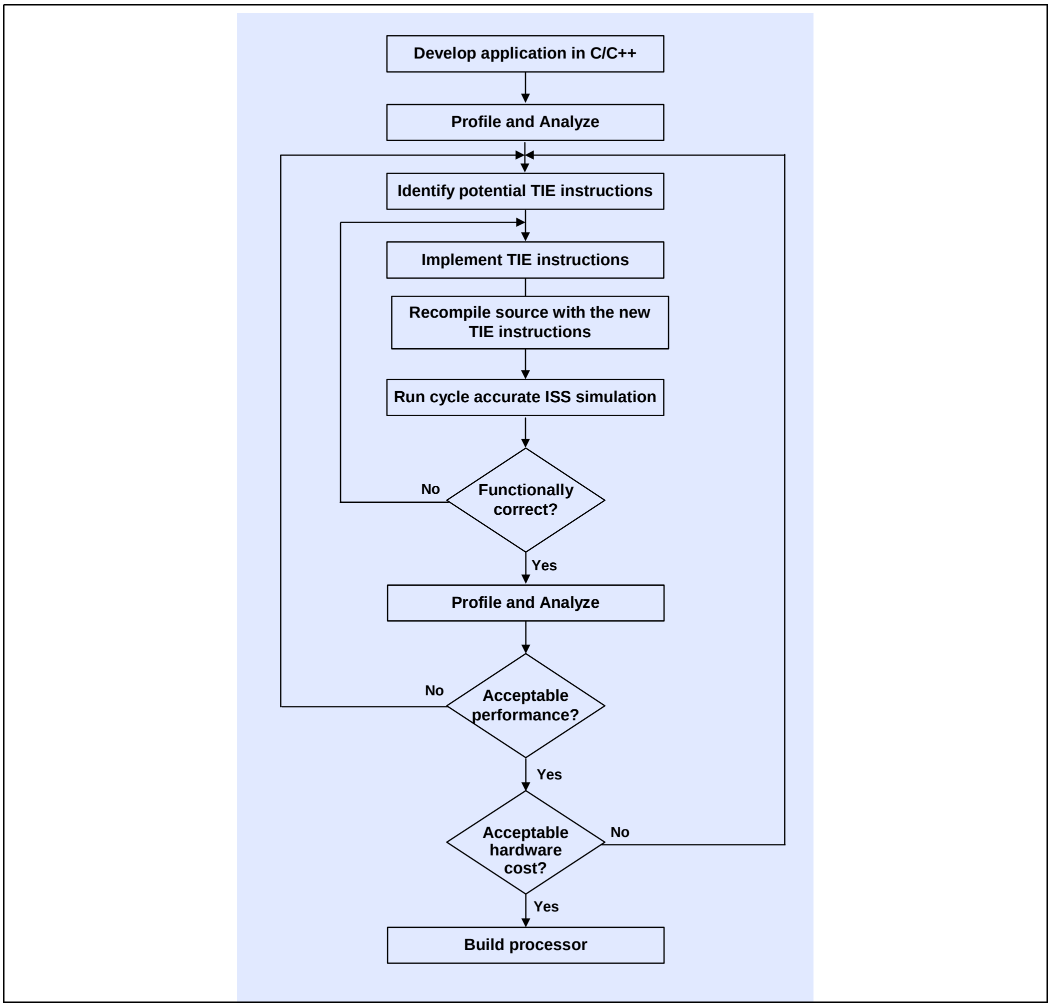

Develop application in C/C++
Profile and Analyze
Identify potential TIE instructions
Implement TIE instructions
Recompile source with the new TIE instructions
Run cycle accurate ISS simulation
Functionally correct?
No
Yes
Profile and Analyze
Acceptable performance?
No
Yes
Acceptable hardware cost?
No
Yes
Build processor

Figure 20. Typical TIE Development Cycle

Typically, a TIE instruction is first implemented with a direct, simple ISA description (instructions plus registers) using the TIE *operation* construct. After you are satisfied that the instructions you create are correct and that your performance goals are being met, you can optimize the TIE descriptions for a more efficient hardware implementation and to share hardware with other instructions using the TIE *semantic* construct. The hardware implementation can be verified against the simple ISA implementation using formal techniques, if desired.

7.3 Improving Application Performance Using TIE

TIE allows you to improve the performance of your application by enabling you to create one TIE instruction that does the work of multiple instructions of a general-purpose processor. There are several different techniques that you can use to combine multiple operations into one. Three common techniques available through TIE are fusion, SIMD/vector transformation, and FLIX. To illustrate these three techniques, consider a simple example where profiling indicates that most of your execution time is spent in the following loop, computing the average of two arrays.

```
unsigned short *a, *b, *c;
...
for (i=0; i<n; i++)
     c[i] = (a[i] + b[i]) >> 1;
```

7.3.1 Fusion

In the above piece of C code, you can see that in every iteration of the loop, two *short* data items are being added together and the result is being shifted right by one bit. Two Xtensa LX core instructions are required for the computation, not counting the instructions required for loading and storing the data. These two operations can be fused into a single TIE instruction, which can be fully described as follows.

```
operation AVERAGE{out AR res, in AR input0, in AR input1} {} {
     wire [16:0] tmp = input0[15:0] + input1[15:0];
     assign res = tmp[16:1];
}
```

The instruction takes two input values (`input0` and `input1`) from the Xtensa core AR register file to compute the output value (`res`). The result is then put into the AR register file. The semantics of the instruction, an add feeding a shift, are described using standard Verilog-like syntax. To use the instruction in C or C++, simply modify the original example code as follows.

```
#include <xtensa/tie/average.h>
unsigned short *a, *b, *c;
...
for (i=0; i<n; i++)
     c[i] = AVERAGE(a[i], b[i]);
```

The AVERAGE instruction can also be used directly in assembly code. The entire software toolchain recognizes AVERAGE as a valid instruction for this processor. The TIE compiler implements this instruction in a manner consistent with the Xtensa processor's pipeline. Thus, the TIE compiler automates the task of implementing instruction extensions, both in hardware as well as in software. This allows the software developer to focus on the semantics of their instruction extensions, without worrying about the implementation details.

Fused instructions are not necessarily as expensive in hardware as the sum of their constituent parts. Often they are significantly cheaper because they operate on restricted data sets. In this example, the addition inside `AVERAGE` is just a 16-bit addition, while the Xtensa instruction set implements 32-bit additions. Further, because the data is shifted by a fixed amount (right shift by 1), the shift operation is essentially free of hardware cost because the fixed 1-bit shift operation can be performed by simply extracting the appropriate bits from the result of the addition.

The `AVERAGE` instruction requires very few gates and easily executes in one cycle. However, TIE instructions need not execute in a single cycle. The TIE *schedule* construct allows you to create instructions with computations that span multiple clock cycles. Such instructions can be fully pipelined (multiple instances of the instruction can be issued back to back) or they may be iterative. Fully pipelined instructions may achieve higher performance because they can be issued back-to-back, but they may also require extra implementation hardware. An iterative instruction spanning multiple clock cycles uses the same set of hardware, over two or more clock cycles. This design approach saves hardware but if the application attempts to issue back-to-back iterative instructions during adjacent cycles, the processor will stall until the first dispatch of the instruction completes its computation (stops using the shared resource).

7.3.2 SIMD/Vector Transformation

In the example of fusion shown above, consecutive add and shift operations are combined into one instruction. Other types of instruction combinations can also improve performance. For example, in every iteration of the loop, the example code performs the same computation on new data. You can use SIMD (single-instruction, multiple-data) instructions (also called vector instructions) to compute multiple iterations of a computation using one instruction. You can also combine fusion and SIMD techniques. Consider, for example, the case where you would like to compute four averages in one instruction. This can be done using the following TIE description:

```
regfile VEC 64 8 v

operation VAVERAGE{out VEC res, in VEC input0, in VEC input1} {} {
     wire [67:0] tmp = {input0[63:48] + input1[63:48],
                   input0[47:32] + input1[47:32],
                   input0[31:16] + input1[31:16],
                   input0[15:0] + input1{15:0]};
     assign res = {tmp[67:52], tmp[50:35], tmp[33:18], tmp[16:1]};
}
```

Note that computing four 16-bit averages requires that each data vector be 64 bits (4x 16 bits) and the Xtensa AR register file is only 32 bits. Therefore, we create a new register file, called VEC, with eight 64-bit registers to hold the 64-bit data vectors for the new SIMD instruction. This new instruction, VAVERAGE, takes two operands from the VEC register file, computes four averages with the 1-bit shift, and saves the 64-bit vector result in the VEC register file. To use the instruction in C/C++, simply modify the original example as follows.

```
#include <xtensa/tie/average.h>
VEC *a, *b, *c;
...
for (i=0; i<n; i+=4) {
     c[i] = VAVERAGE(a[i], b[i]);
```

The automatically generated Xtensa LX compiler recognizes a new 64-bit C/C++ datatype VEC, corresponding to the new register file. In addition to VAVERAGE, the TIE compiler automatically creates new load and store instructions to move 64-bit vectors between the VEC register file and memory. The C compiler uses these instructions to load and store the 64-bit arrays of type VEC.

Note that the Xtensa processor must be configured with a PIF (the main processor bus) width of at least 64 bits to move 64-bit data between external memory and the new 64-bit VEC register file.

Compared to the fused instruction AVERAGE, the vector-fused instruction VAVERAGE requires significantly more hardware because it performs four 16-bit additions in parallel (in addition to the four 1-bit shifts, which do not require any additional gates). However, the performance improvement from combining vectorization and fusion is significantly larger than the performance improvement from fusion alone.

7.3.3 FLIX

A third technique to perform multiple operations in a single instruction is to use the Flexible Length Instruction Xtensions (FLIX) capabilities of the Xtensa LX processor. Fusion allows you to combine dependent operations in a single instruction. SIMD/vector transformation allows you to replicate the same operation multiple times in one instruction. In contrast, FLIX allows you to combine *unrelated* operations in one instruction word.

A SIMD/vector operation has no more operands than the original scalar instruction. Because intermediate results and operands in fused instruction never go into or out of the register file, a fused instruction typically has a similar number of operands as the original instructions being fused. In contrast, when combining unrelated operations into one instruction word, the number of operands grows in proportion to the number of operations being combined. If many or all of these operands reside in one register file, this operand growth caused by the increase in the number of parallel operations may affect the number of read and write ports required on that register file.

Xtensa LX core instructions are either 16 or 24 bits wide. It is very difficult to combine even two unrelated operations into one instruction and encode all the required operands in just 24 bits. Therefore, in addition to 24-bit TIE instructions, the Xtensa LX processor allows TIE instructions to be either 32 or 64 bits wide. Instructions of different sizes can be freely intermixed without setting any “mode bit.” In fact, the Xtensa LX processor has no mode setting for instruction size. The instruction size is encoded into the instruction word itself. The Xtensa LX processor identifies, decodes, and executes any mix of 16-, 24-, and 32- or 64-bit instructions from the incoming instruction stream. The 32-bit or 64-bit instruction can be divided into slots, with independent operations placed in either, all, or some of the slots. The slots need not be equally sized, which is why this feature is called Flexible Length Instruction Xtensions (FLIX). Any combination of the operations allowed in each slot can occupy a single FLIX instruction word.

Consider again our `AVERAGE` example. On a base Xtensa LX processor, the inner loop would contain the `ADD` and `SRAI` instruction to do the actual computation, and two `L16I` load instructions and one `S16I` store instruction as well as three `ADDI` instruction to update the address pointers used by the loads and stores. To accelerate this code, you could create a 64-bit FLIX instruction format with one slot for the load and store instructions, one slot for the computation instructions, and one slot for address-update instructions. This can be done using the following TIE description:

```
format flix3 64 {slot0, slot1, slot2}

slot_opcodes slot0 {L16I, S16I}
slot_opcodes slot1 {ADDI}
slot_opcodes slot2 {ADD, SRAI}
```

The first declaration creates a 64-bit instruction and define an instruction format with three opcode slots. The last three lines of code list the instructions available in each opcode slot defined for this FLIX configuration. Note that all the instructions specified are predefined, core processor instructions, so their definition need not be provided in the TIE code, because the TIE compiler already knows about all core Xtensa LX instructions.

For this example, no changes are needed to the C/C++ program. The generated C/C++ compiler will compile the original source code, taking advantage of FLIX automatically. For example, the generated assembly code for this processor implementation would look like this:

```
loop:
{addi a9,a9,4;      add  a12,a10,a8;    l16i a8,a9,0   }
{addi a11,a11,4;    srai a12,a12,1;     l16i a10,a11,0 }
{addi a13,a13,4;    nop;                s16i a12,a13,0 }
```

Thus, a computation that would have taken eight cycles per iteration on a base Xtensa LX processor is now accomplished in just 3 cycles per iteration. It took only five lines of TIE code to specify a complex FLIX configuration format with three instruction slots using existing Xtensa LX opcodes, as shown in Table 68.

Table 68. Performance Tuning with Fusion, SIMD, and FLIX Instruction Extensions

	Number of Cycles Per Iteration of "AVERAGE" Function
Base Processor	8
Processor with Fused Instructions	7
Processor with SIMD Instructions	7
Processor with FLIX Instructions	3

This simple example illustrates how to replicate Xtensa LX core instructions into multiple instruction slots. It is equally easy to create a FLIX configuration with slots that contain a mix of Xtensa core instructions and designer-defined TIE instructions.

FLIX has several advantages compared to the other techniques. No changes to the C/C++ source are required when you use FLIX to replicate Xtensa core instructions in multiple instruction slots. A particular FLIX implementation is more likely to apply to a wider range of C/C++ programs than a TIE implementation using fusion or SIMD/vector transformation.

However, the other techniques also have advantages. Fusion often adds just a small amount of hardware. SIMD instructions tend to give larger performance improvements. Which technique you use depends on your goals, budget, and application. Of course, all the techniques can be combined. Because Tensilica's processor generator is very fast, it's easy to experiment with different approaches to find the fastest, most efficient Xtensa LX processor architecture for a target application.

Note that even without FLIX and its 32- or 64-bit instructions, it is possible to combine independent operations into a single instruction. It is *not* possible to fit two operations, each referring to three register operands, into a 24-bit instruction because of the number of bits needed to encode the register-file-address operands. However, TIE allows you to create STATE registers that are implicitly addressed so they do not take any instruction-word bits for operand-source or result-destination encoding. Using the generated assembly from the previous FLIX example as a template, you could create three separate

TIE instructions operating on six pieces of state, labelled `state8` through `state13`. The first instruction, for example, would increment `state9` by four, set `state12` to the sum of `state10` and `state8`, and load from the contents of `state9` into `state8`. Because each instruction always reads or writes a specific set of states rather than one of many register-file entries, not many bits are required to encode the instruction. A full TIE description of this example appears below:

```
state state8 16 add_read_write
state state9 32 add_read_write
state state10 16 add_read_write
state state11 32 add_read_write
state state12 16 add_read_write
state state13 32 add_read_write

operation inst1{}{inout state9, out state12, in state10, inout state8,
                    out VAddr, in MemDataIn32} {
     assign VAddr = state9;
     assign state9 = state9 + 32’d4;
     assign state12 = state10 + state8;
     assign state8 = MemDataIn32;
}
operation inst2{}{inout state11, inout state12, out state10,
                    out VAddr, in MemDataIn32} {
     assign VAddr = state11;
     assign state11 = state11 + 32’d4;
     assign state12 = {1’d0, state12[15:1]};
     assign state10 = MemDataIn32;
}

operation inst3{}{inout state13, in state12,
               out VAddr, out MemDataOut32} {
     assign VAddr = state13;
     assign state13 = state13 + 32’d4;
     assign MemDataOut32 = state12;
}
```

In addition to using TIE state, this example requires custom load and store instructions. By setting the `VAddr` interface, the core load/store unit is instructed to perform a load or store operation from or to the programmer-defined address. The data from a load operation is returned to the TIE execution unit via the `MemDataIn32` interface. Data to be stored is assigned via the TIE execution unit to the `MemDataOut32` interface. When a TIE state is declared with the attribute *add_read_write*, the TIE compiler automatically creates instructions to move the state to and from the general-purpose AR register file. The read instruction has the mnemonic `RUR.<state name>` and the write instruction has the mnemonic `WUR.<state name>`.

To use these instructions in C/C++, modify the original example as follows.

```
#include <xtensa/tie/average.h>
unsigned short *a, *b, *c;
...
WUR.state9(a);
WUR.state11(b);
WUR.state13(c);
for (i=0; i<n; i++) {
     inst1();
     inst2();
     inst3();
}
```

The first three instructions initialize the state registers to the addresses of the arrays. Inside the loop, the three instructions cycle through the computation. Note that none of these instructions take any arguments because they operate only on TIE state, and they allocate no bits to specify a register-file address or immediate operands.

This style of combining independent operations can provide performance gains similar to FLIX with more modest hardware costs. However, this type of TIE tends to be less flexible and more difficult to program. If a different application needs to perform adds, shifts, and loads in parallel, but data for these operations comes from or results go to different states, the instructions created by the original TIE would not be general enough to meet these requirements. The FLIX approach is more appropriate in this situation.

7.4 TIE Queues and Ports

The TIE features described so far communicate to the world outside of the processor only through the processor's memory and PIF bus interfaces. In many embedded applications, the processor must communicate with other devices in the system in addition to the memories. It might also need access to local storage at a much wider bandwidth than is provided by the processor's existing memory interfaces. The system design may require direct processor-to-processor communications without the overhead of shared memory. The TIE language provides two convenient and flexible ways for the processor to communicate with one or more external devices: TIE queues and ports.

A queue is commonly used to communicate between different devices in a System-On-Chip (SOC) design. The TIE language provides constructs that allow the Xtensa LX processor to directly interface with input or output queues. Designer-defined instructions can "pop" an entry from an input queue and use the incoming data for a computation or for flow control. Similarly, the results of a computation (or alternately, a command packet for another device) can be "pushed" onto an output queue. After a queue is defined in TIE, the TIE compiler automatically generates all the interface signals for the additional port needed to connect to the queue. The logic to stall the processor when it wants to read an empty input queue or write to a full output queue is also automatically generated.

The TIE language for the Xtensa LX processor also supports an “import-wire” construct that the processor uses to directly sample the value of an external signal and an “export-state” construct that the processor uses to drive the value of any TIE state on external signals. These input and output signals are collectively called TIE ports.

Note: For details on these features, see Chapter 15. System design aspects of TIE ports and queues are also described in Chapter 16.

8. Performance Programming Tips for the Software Developer

Through TIE, Tensilica gives you the ability to tailor your processor to your application, potentially giving you performance levels not possible with traditional processors. However, even without TIE, there is much you can do to help maximize your application's performance, and Tensilica provides several tools to help. Furthermore, cleaning and tuning your code can often make it even more amenable to TIE acceleration, particularly when using TIE through automated tools such as the XPRES compiler.

8.1 Diagnosing Performance Problems

Without tools, it can be difficult to know where your performance problems are or what you can do to solve them. For example, a floating-point multiplication using software emulation might take approximately 100 cycles. If 1% of the operations in your application are floating-point multiplies, the multiplication routine represents significant computational overhead. If 0.01% of the operations are floating-point multiplies, the overhead might be negligible.

There are two classes of tools, both available through simulation, that can help you diagnose software-performance problems. Profiling, available through Xtensa Xplorer as well as through xt-gprof, shows you where your application code is spending the most time, either at the function level or at the source-line or assembly-line level. You should of course concentrate your efforts on regions that consume a significant portion of the processor's or task's time. In addition, the simulator is able to break down the time spent executing an application event-by-event. For example, the simulator can tell you how much time is spent globally servicing cache misses or accessing uncached memory. This information helps you know whether to concentrate your efforts on generated code sequences or on your memory subsystem. Profiling and simulation data can also be combined. You can profile code based on cache misses, for example, to see which portions of a program cause the largest number of cache misses.

8.2 Types of Performance Problems

When you know where to start, you can begin to tune your code. There are several aspects to performance including algorithmic, configuration, memory system, microarchitectural, and compiler code quality. We will discuss each in turn.

8.3 Choosing Algorithms

Choice of algorithm has a first-order effect on performance. A radix-4 FFT fundamentally has fewer multiplies than a radix-2 FFT. A heap sort fundamentally requires fewer compares than an insertion sort. Algorithm choice is mostly independent of processor architecture, and there are few tools to help select among alternative algorithms. However, there is some connection between algorithm and processor or system architecture.

For example, a radix-4 FFT requires fewer multiplies than a radix-2, and a radix-8 FFT requires fewer multiplies than either of the other two types. However, as you increase the FFT radix, you increase the operation complexity and the number of registers required. At sufficiently high radix, performance becomes dominated by overhead and data shuffling. A typical Xtensa LX processor configuration tends to handle radix-4 better than radix-8. Be mindful of your architecture when choosing algorithms and, for the Xtensa LX processor, be mindful of your algorithm when tuning the architecture.

8.4 Configuration

The choice of processor configuration can greatly affect performance. As previously described, the use of TIE can increase performance by many factors, but configuration parameters and choice of memory subsystem can also make a substantial difference.

There are two basic techniques that can be used to improve your choice of core processor configuration. Using the first technique, profile your application and take a closer look at the hotspots. If profiling shows, for example, that much time is spent in integer multiplication emulation, adding a multiplier to your configuration will probably help. Using the second technique, build multiple configurations and see how performance of your application varies across configurations. Xtensa Xplorer allows you to manage multiple configurations and graphically compare performance of your application using different configurations.

In Figure 21, we use Xtensa Xplorer to compare performance of an application performing floating-point multiplies using two configurations; one with the Xtensa LX floating-point unit (FPU) and one without. The figure shows two sets of bars. The left-hand set of three bars shows software cycle counts for a processor configuration with an FPU. The right-hand set of bars shows a processor configuration without an FPU. Within each set of three bars, the first bar shows the total cycle count for the software routine, the second bar shows the branch delays, and the third bar shows the interlock cycles. As shown in Figure 21, the floating-point unit greatly improves performance.

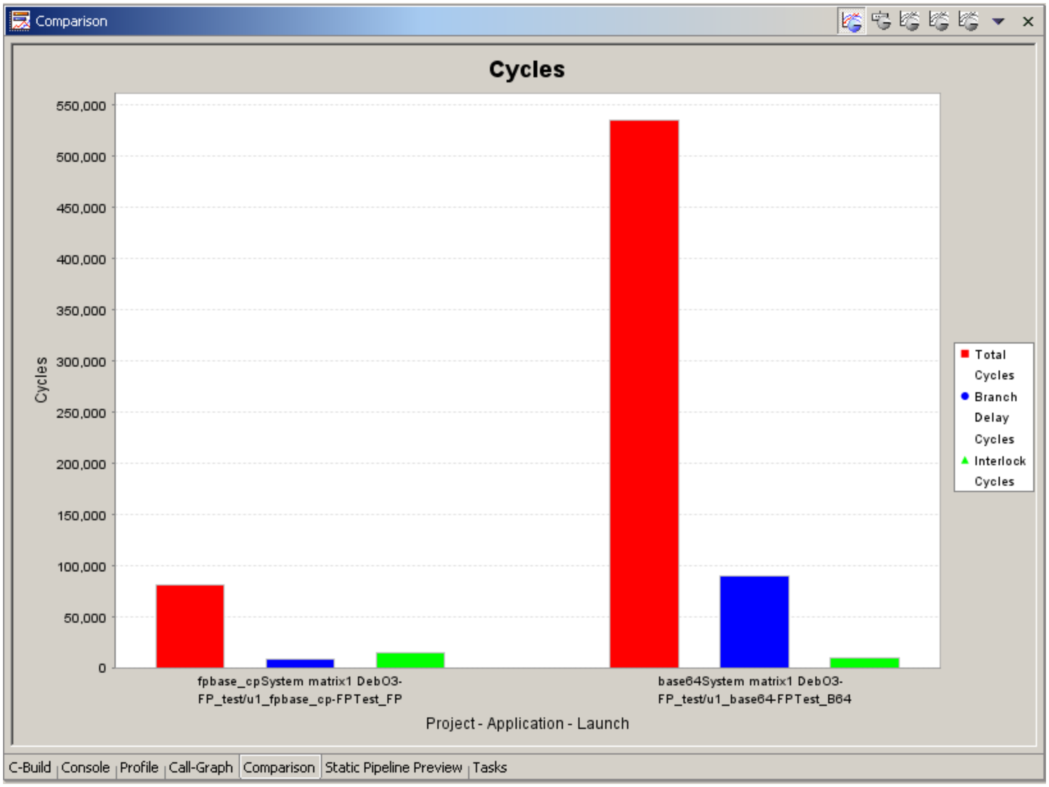


Figure 21. Adding a Floating-Point Unit to a Configuration

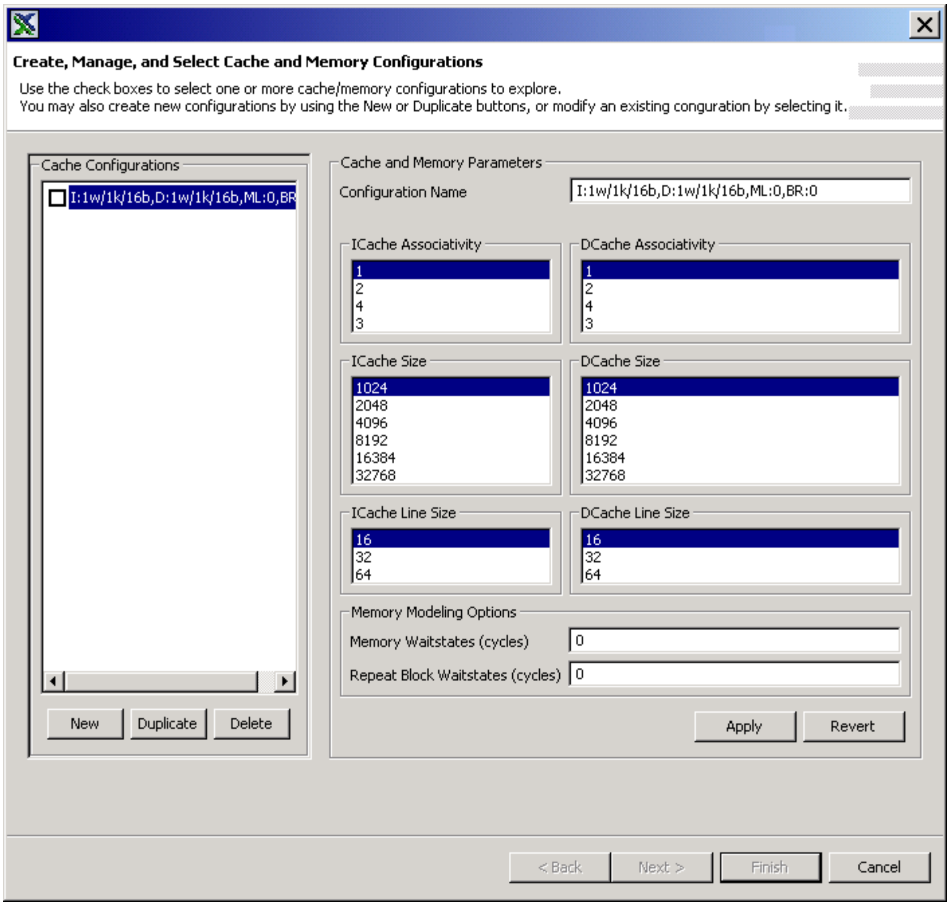
8.5 Memory System

In modern embedded systems, how fast you get data into and out of the processor can often be the most critical aspect of performance. The Xtensa LX processor gives you a lot of freedom in designing external interfaces. Communication can be through local memory ports, the Xtensa Local Memory Interface (XLMI), the PIF, caches, TIE queues and ports, or interrupt lines. Each of these Xtensa LX interface types can be tailored for the application. You can, for example, choose the width of the PIF, the size of caches (if any), and the number of memory units.

Tensilica provides several tools to measure and analyze memory-system performance. The standalone ISS simulator accurately models caches and the interactions between the memory system and the pipeline for a fixed-latency/fixed-bandwidth memory system. XTMP allows you to accurately model more complicated memory systems including DMA, TIE queues and ports, and custom devices and interconnects.

The profiler, either through xt-gprof or through Xtensa Xplorer, allows you to profile your application code based on cache misses, allowing you to see which regions of your code suffer the most cache misses or spend the most time handling cache misses.

As with other aspects of configuration, Xtensa Xplorer allows you to easily compare performance of an application using multiple memory systems. In addition, the Cache Explorer feature of Xtensa Xplorer, shown in Figure 22, allows you to graphically select a range of cache parameters and have the system automatically simulate and compare performance for all selected configurations.


Create, Manage, and Select Cache and Memory Configurations

Use the check boxes to select one or more cache/memory configurations to explore.
You may also create new configurations by using the New or Duplicate buttons, or modify an existing conguration by selecting it.

Cache Configurations

☐ I:1w/1k/16b,D:1w/1k/16b,ML:0,BR

Cache and Memory Parameters

Configuration Name: I:1w/1k/16b,D:1w/1k/16b,ML:0,BR:0

ICache Associativity: 1 (selected), 2, 4, 3

DCache Associativity: 1 (selected), 2, 4, 3

ICache Size: 1024 (selected), 2048, 4096, 8192, 16384, 32768

DCache Size: 1024 (selected), 2048, 4096, 8192, 16384, 32768

ICache Line Size: 16 (selected), 32, 64

DCache Line Size: 16 (selected), 32, 64

Memory Modeling Options

Memory Waitstates (cycles): 0

Repeat Block Waitstates (cycles): 0

Figure 22. Cache Explorer

When tuning your memory subsystem, it is important to model a system and application similar to the one you actually will use. For example, on a system with local memories, DMA might significantly improve performance for some applications. Simulating such applications without modeling DMA might lead one to incorrect conclusions about the required cache parameters.

By default, the stand-alone ISS does not simulate the memory subsystem; it assumes zero delay cycles on all loads and stores. This policy allows you to tune the other aspects of performance independently of your memory subsystem. Particularly with caches, small changes that improve microarchitectural performance might randomly hurt memory-system performance as they change data or instruction layout. Nonetheless, all these small changes in aggregate will probably improve performance. Simulations with an ideal memory system allow you to ignore these effects until you are ready to tune your memory system. After architectural tuning of the processor is finished, you can add a memory model to the stand-alone ISS simulation, or use the XTMP simulation environment to add non-zero memory-delay characteristics to the simulation.

8.6 Microarchitectural Description

At full speed, the Xtensa LX pipeline can execute one instruction every cycle. However, various situations can stall the pipeline. The most common reasons for stalling the processor are memory delays, branch delays, and pipeline interlocks. The ISS summarizes the number of cycles stalled for different reasons. An example output is shown in Figure 23.

```
Cycles: total = 17893893
                                                     Summed |            Summed
                                             CPI      CPI   |% Cycle   % Cycle
Committed instructions        15313272 ( 1.0000   1.0000 |  85.58     85.58 )
Taken branches                 1948514 ( 0.1272   1.1272 |  10.89     96.47 )
Source interlocks               106274 ( 0.0069   1.1342 |   0.59     97.06 )
Icache misses                   480983 ( 0.0314   1.1656 |   2.69     99.75 )
Dcache misses                     5246 ( 0.0003   1.1659 |   0.03     99.78 )
Exceptions                         835 ( 0.0001   1.1660 |   0.00     99.78 )
Uncached ifetches                 7493 ( 0.0005   1.1665 |   0.04     99.83 )
Uncached loads                      31 ( 0.0000   1.1665 |   0.00     99.83 )
Sync replays                      3696 ( 0.0002   1.1667 |   0.02     99.85 )
Special instructions               415 ( 0.0000   1.1667 |   0.00     99.85 )
Loop start overhead              27129 ( 0.0018   1.1685 |   0.15    100.00 )
Reset                                5 ( 0.0000   1.1685 |   0.00    100.00 )
```

Figure 23. Example Pipeline Delays

In addition, the profiler allows you to create profiles based on events, such as branch penalties. This feature allows you, for example, to identify functions, lines, or instructions in your program that suffer the most from branch penalties.

Because Xtensa LX pipeline delays are small, pipeline interlocks often do not seriously degrade performance. Interlocks that do exist are often hard to avoid. As in the example in Figure 24, branch penalties are often larger. Except for zero-overhead loops, every aligned branch taken in the 5-stage version of the Xtensa LX pipeline suffers a 2-cycle branch penalty. One additional penalty cycle is required for unaligned branch targets (the instruction at the branch target address is not aligned so that the entire instruction fits in a single, aligned-fetch boundary, 32 or 64 bits depending on processor configuration). A 7-stage version of the Xtensa LX pipeline adds one additional branch-penalty cycle.

Note: If the branch is not taken, there are no branch-penalty cycles.

The Xtensa C/C++ compiler (XCC) reduces branch penalties in two different ways. First, it inlines functions into their caller to avoid call and return branch delays. Note that when compiling with only the -O2 or -O3 compiler flags, the compiler can only inline functions when both the caller and the callee are in the same compilation unit or file. If you compile with the `-ipa` (interprocedural analysis) compiler flag, the compiler can inline any direct call regardless of file boundaries. Second, the compiler tries to lay out code so that the most frequent code traces become straight lines. Consider the example in Figure 24.

```
Statement 1;            Statement 1             Statement 1
if (cond) {             cond branch to L        cond branch to L
  Statement 2;          Statement 2             L3: Statement 3
}                       L: Statement 3          ...
Statement 3;                                    L: Statement 2
                                                jump to L3:
```

Figure 24. Branch Directions

The compiler can choose to generate code that conditionally jumps around *Statement 2,* as shown in the center example. Alternatively, the compiler can generate code that jumps to someplace else that contains *Statement 2* followed by a jump back to *Statement 3* as shown in the rightmost example. The code in the center example will suffer a taken branch penalty whenever *cond* is not true but no penalty whenever *cond* is true. The code in the rightmost example will suffer two taken branch penalties whenever *cond* is true but none whenever *cond* is not true. The best choice depends on how often *cond* is true during program execution.

By default, the compiler guesses branch frequencies. You can help the compiler in two ways. First, the compiler supports a pragma, *#pragma frequency_hint,* with values of *FREQUENT* or *NEVER* to that tell the compiler when you know that a branch is very likely or very unlikely. Second, the compiler accepts feedback from the profiler, which gives it actual run-time statistics about all code branches.

8.7 Compiled Code Quality

Looking at your code, you often have expectations about the code that the compiler will generate. When tuning code, it is often helpful to take a close look at generated assembly code to see if it is as tight as you expect. There are many reasons why code quality might not be as good as expected. We will discuss some of the more common ones.

8.7.1 Choosing Compiler Flags

The compiler has many options to control the amount and types of optimizations it performs. Choosing the right options can significantly increase performance.

By default, the compiler compiles a file with *-O0* (no optimization). The compiler makes no effort to optimize the code either for space or performance. The main goals for unoptimized code generation are fast compilation and ease of debugging.

The lowest-level compiler optimization flag is *-O2*. The *-O3* compiler flag enables additional optimizations that are mostly (but not exclusively) useful for DSP-like code. Orthogonal to the use of *-O2* or *-O3*, the *-Os* tells the compiler to optimize for space rather than speed. The flag *-Os* can be used with either *-O2* or *-O3*; if neither is given, the compiler assumes *-O2*.

By default, the compiler optimizes one file at a time. The compiler is also capable of performing interprocedural analysis, which optimizes the entire application program across multiple source files. To use interprocedural analysis, add the `-ipa` flag to both the compiler and linker command lines. Behind the scenes, the compiler will delay optimization until the link step. The `-ipa` option can be used with or without *-Os*. The *-Os* works very well together with `-ipa` because interprocedural analysis allows the compiler to delete functions that are never used.

The compiler can make use of profile data during the compilation process through use of the compiler flags *-fb_create* and *-fb_opt*. As mentioned in Section 8.6, the use of feedback enables the compiler to mitigate branch delays. In addition, the use of feedback allows the compiler to inline just the more frequently called functions and to place register spills in infrequent regions of code. The use of feedback allows the compiler to optimize for speed in the important portions of your application while optimizing for space everywhere else.

8.7.2 Aliasing

Consider the code in Figure 25. You might assume that the compiler will generate code that loads **a* into a register before the loop and contains an inner loop that loads *b[i]* into a register and adds it into the register containing **a* in every iteration.

```
void foo(int *a, int *b)
{
  int i;
  for (i=0; i<100; i++) {
   *a += b[i];
  }
}
```

Figure 25. Aliased Parameters

In fact, you will find that the compiler generates code that stores **a* into memory on every iteration. The reason is that *a* and *b* might be aliased; **a* might be one element in the *b* array. While it is very unlikely in this example that the variables will be aliased, the compiler cannot be sure.

There are several techniques to help the compiler optimize better in the presence of aliasing. You can compile using *-ipa*, you can use globals instead of parameters, you can compile with special flags, or you can annotate a variable declaration with the __*restrict* attribute.

8.7.3 Short Datatypes

Most Xtensa instructions operate on 32-bit data. There are no 16-bit addition instructions. The system emulates 16-bit addition using 32-bit arithmetic and this sometimes forces the system to extend the sign of a 16-bit result to the upper bits of a register. Consider the example in Figure 26. You might expect that the routine would use a single add instruction to add *a* and *b* and place the result in the return register. However, C semantics say that if the result of *a+b* overflows 16 bits, the compiler must set *c* to the sign-extended value of the result's lower 16 bits, leading to the code in Figure 27.

```
int foo(short a, short b)
{
 short c;
 c = a+b;
 return c;
}
```

Figure 26. Short Temporaries

```
entry a1,32
add.n a2, a2, a3
slli a2, a2, 16
srai a2, a2, 16
retw.n
```

Figure 27. Short Temporaries Assembly File

Much more efficient code would be generated if *c* was declared to be an *int*. In general, avoid using *short* and *char* for temporaries and loop variables. Try to limit their use to memory-resident data. An exception to this rule is multiplication. If you have 16-bit multipliers and no 32-bit multipliers, make sure that the multiplicands' data types are *short*.

8.7.4 Use of Globals Instead of Locals

Global variables carry their values throughout the life of a program. The compiler must assume that the value of a global might be used by calls or by pointer dereferences. Consider the example in Figure 28.

```
int g;
void foo()
{
 int i;
 for (i=0; i<100; i++){
  fred(i,g);
 }
}
```

Figure 28. Globals

Ideally, g would be loaded once outside of the loop, and its value would be passed in a register into the function `fred`*.* However, the compiler does not know that `fred` does not modify the value of g. If `fred` does not modify g, you should rewrite the code using a local variable as shown in Figure 29. Doing so saves a load of g into a register on every loop iteration.

```
int g;
void foo()
{
 int i, local_g=g;
 for (i=0; i<100; i++){
  fred(i,local_g);
 }
}
```

Figure 29. Replacing Globals with Locals

8.7.5 Use of Pointers Instead of Arrays

Consider a piece of code that accesses an array through a pointer such as in Figure 30.

```
for (i=0; i<100; i++)
  *p++ = ...
```

Figure 30. Pointers for Arrays

In every iteration of the loop, **p* is being assigned, but so is the pointer *p*. Depending on circumstances, the assignment to the pointer can hinder optimization. In some cases it is possible that the assignment to **p* changes the value of the pointer itself, forcing the compiler to generate code to reload and increment the pointer during each iteration. In other cases, the compiler cannot prove that the pointer is not used outside the loop, and the compiler must therefore generate code after the loop to update the pointer with its incremented value. To avoid these types of problems, it is better to use arrays rather than pointers as shown in Figure 31.

```
for (i=0; i<100; i++)
  p[i] = ...
```

Figure 31. Arrays Instead of Pointers

9. Hardware Overview and Block Diagram

Figure 32 shows a block diagram of the Xtensa LX microprocessor with its major blocks and its external connections. Most blocks have a number of configurable options allowing the Xtensa LX core to be tailored to a specific application. The type and size of local instruction and data memories, as well as the Xtensa Processor Interface (PIF) are also configurable. The designer can make application-specific extensions to the processors defining new register files and instructions using the TIE language. The following sections give details on the different configuration options, as well as details on the interfaces and protocols between the Xtensa LX core and the outside world.

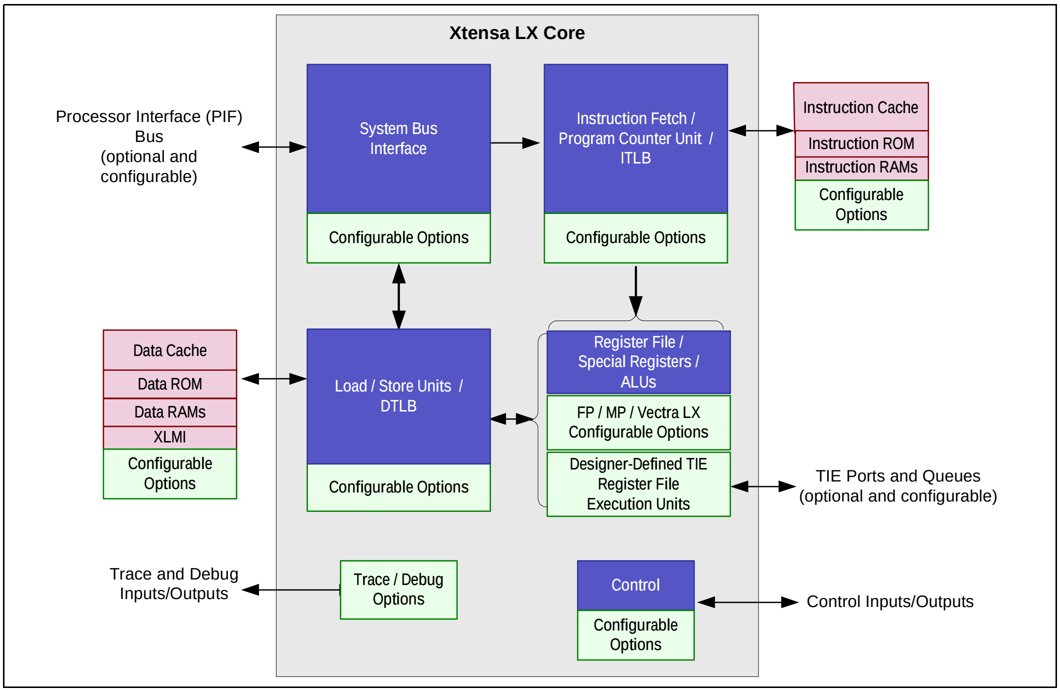

Xtensa LX Core
Processor Interface (PIF) Bus (optional and configurable)
System Bus Interface
Configurable Options
Instruction Fetch / Program Counter Unit / ITLB
Configurable Options
Instruction Cache
Instruction ROM
Instruction RAMs
Configurable Options
Data Cache
Data ROM
Data RAMs
XLMI
Configurable Options
Load / Store Units / DTLB
Configurable Options
Register File / Special Registers / ALUs
FP / MP / Vectra LX Configurable Options
Designer-Defined TIE Register File Execution Units
TIE Ports and Queues (optional and configurable)
Trace and Debug Inputs/Outputs
Trace / Debug Options
Control
Configurable Options
Control Inputs/Outputs

Figure 32. Configurable and Extensible Xtensa System

9.1 *The Xtensa LX CPU*

The Xtensa LX processor core consists of a number of main blocks, including the core register files and execution units, the load/store units, the program-counter unit (PC unit) and instruction-fetch unit, the external interface unit, trace and debug units, and a processor-control unit. Each unit offers a number of options that allow SOC designers to tailor them for a specific application and to meet area, power, and performance goals.

The register files and execution unit together implement the processor’s general-purpose AR registers, special processor registers, and most of the instructions in the core instruction set architecture (ISA). The AR register file can be configured to have either 32 or 64 registers. The Xtensa ISA employs a technique called “register windowing” to create a compact, efficient instruction-encoding scheme while maintaining the good execution performance that results from having a large available register file. The AR register file typically has two read ports and one write port, but additional read and write ports may be added if required for designer-defined TIE (Tensilica Instruction Extension language) extensions. The set of special registers present in a specific instance of the Xtensa LX processor core depends on the configuration options chosen.

Additional processor instructions can be added to the core architecture by including one of a number of predefined coprocessors and execution units (see Section 10.2), or designer-defined coprocessors and execution units using the TIE language. Optional and designer-defined coprocessors and execution units can use the existing general-purpose AR register file and can also define and use special, dedicated register files.

Load/store units implement the Xtensa LX load and store instructions. An Xtensa LX processor can have one or two load/store units. All core processor instructions use the first load/store unit. The optional second load/store unit is used by TIE-based instruction extensions. The second load/store unit can also be used by the inbound-PIF feature, which allows core processor instructions to perform loads and stores at the same time that inbound-PIF (DMA) transfers are occurring.

The load/store units connect to one of four types of local data memories. These local memories can include any of those listed in Table 69:

Table 69. Xtensa LX Data-Memory Port Types and Quantities

Data-Memory Port Type	Minimum Number per Core	Maximum Number Per Core	Description
Data Cache Ways	0	4	Data caches may be direct-mapped or have 2, 3, or 4 ways
Data RAM	0	2	Local data read/write memories
Data ROM	0	1	Local data read-only memory
XLMI Port	0	1	High-speed local data/memory port
All		4	Total, all types of data memory ports

If a data cache is configured, the load/store unit manages the data cache based on either a write-through or write-back policy. The Xtensa LX processor core can be configured with one or two load/store units, allowing as many as two load/store operations to occur in one instruction. The optional second load/store unit is only available in conjunction with local-memory options (Data RAM, Data ROM, and the XLMI port) and only with Xtensa LX processor configurations that have no caches. Each load/store unit has a separate interface port to each local memory and both ports must be connected to each configured local memory. (In other words, these ports cannot be left unconnected.) Consequently, each local memory must be designed to interface to two processor ports for Xtensa LX configurations with two load/store units, either through a memory-port arbiter or through the use of a true dual-ported memory design.

The PC and instruction-fetch units control the fetching of instructions. The PC unit provides addresses to as many as four local instruction memories. The number and types of instruction memories are listed in Table 70:

Table 70. Xtensa LX Instruction-Memory Port Types and Quantities

Instruction-Memory Port Type	Minimum Number per Core	Maximum Number Per Core	Description
Instruction Cache Ways	0	4	Instruction caches may be direct-mapped or have 2, 3, or 4 ways
Instruction RAM	0	2	Local instruction read/write memories
Instruction ROM	0	1	Local instruction read-only memory
All		4	Total, all types of instruction-memory port

The instruction-fetch unit controls instruction-cache loading and feeds instructions to the execution unit. The basic Xtensa instruction size is 24-bits, and there is a configuration option that adds a fully compatible set of 16-bit instructions for improved code density. The 16- and 24-bit instructions can be freely intermixed.

FLIX (Flexible-Length Instruction Xtensions) technology makes it possible to add designer-defined 32- or 64-bit instructions to the Xtensa ISA. These wide-word options allow designers to create more complex, compound machine instructions to improve code and application performance. The 32- or 64-bit instructions can be freely intermixed with the 24- and 16-bit Xtensa instructions. (Note that an Xtensa LX configuration may have either 32- or 64-bit FLIX instructions, but not both.)

The external-interface unit provides an interface to the external processor interface bus (PIF). The PIF can be configured to different widths (32, 64, or 128 bits), and consists of separate, unidirectional input and output buses. The external interface unit manages data transfers between the PIF and the processor's local instruction-memory ports, data-memory ports, and the XLMI port. In particular, this unit manages data and instruction cache-line requests and provides inbound-PIF (external PIF master) capabilities to the processor's local instruction and data RAMs.

The control unit controls global operations such as reset, interrupts, and exception prioritization.

Finally, the Xtensa LX processor's On-Chip Debug (OCD) unit can be interfaced to a separate JTAG module (also provided by Tensilica) for chip debug and test. The OCD module provides access to software-visible processor state. In addition, a separate trace port gives visibility into the Xtensa LX processor during real-time program execution.

9.2 Optional ALU Functions and Coprocessors

The Xtensa LX processor provides a number of Coprocessor options, ranging from simple additions of ALU type units and instructions, to full fledged DSP engines, and designer-specified TIE coprocessors. Options include:

- MAC16
  The MAC16 option adds a 16-bit multiplier and a 40-bit accumulator (ACC), eight 16-bit operand registers (or four 32-bit operand registers), load instructions for operand registers, and a set of compound instructions that combine multiply, accumulate, operand load and address update operations. The operand registers can be loaded with pairs of 16-bit values from memory in parallel with MAC16 operations. MAC16 can sustain algorithms with two loads per multiply/accumulate.
- 32-bit Multiply
  The MUL32 option adds to the library of existing MAC16 (16x16 Multiply with 40-bit accumulator) and MUL16 options. This option provides instructions that perform 32x32 multiplication, producing a 64-bit result.
- IEEE754 Floating-point Unit (FPU)
  This 32-bit single precision floating point coprocessor option is optimized for printing, graphics, and audio applications. An IEEE754 compatible FPU, this coprocessor provides the programming ease of floating-point arithmetic with the execution speed of fixed-point processing.
- Vectra LX DSP Engine
  The Vectra LX DSP Engine is optimized to handle high-performance digital signal processing applications using fixed-point arithmetic. As such, this SIMD (single instruction multiple data) DSP option is ideal for communications, audio and imaging applications employing a highly efficient and easy-to-program vector architecture.
- Designer-Defined Coprocessors
  The designer can define a coprocessor using the TIE language. Generic operations common to all coprocessors such as moving data to and from user-defined registers, or branch instructions that branch on the result of a coprocessor operation, are generated for these coprocessors by the Xtensa Processor Generator.

9.3 Local Memories

The Xtensa LX processor has a number of local memory options that allow fast access to instructions and data over a wide range of different applications. The four memory options provided are: the cache interface, the RAM interface, the ROM interface, and the Xtensa Local Memory Interface (XLMI). Local instruction memories can have a total of 4 arrays consisting of zero to four instruction cache ways, zero to two instruction RAMs, and zero or one instruction ROM. Local data memories can have a total of four arrays consisting of zero to four cache ways, zero to two data RAMs, zero or one data ROM, and zero or one XLMI interface.

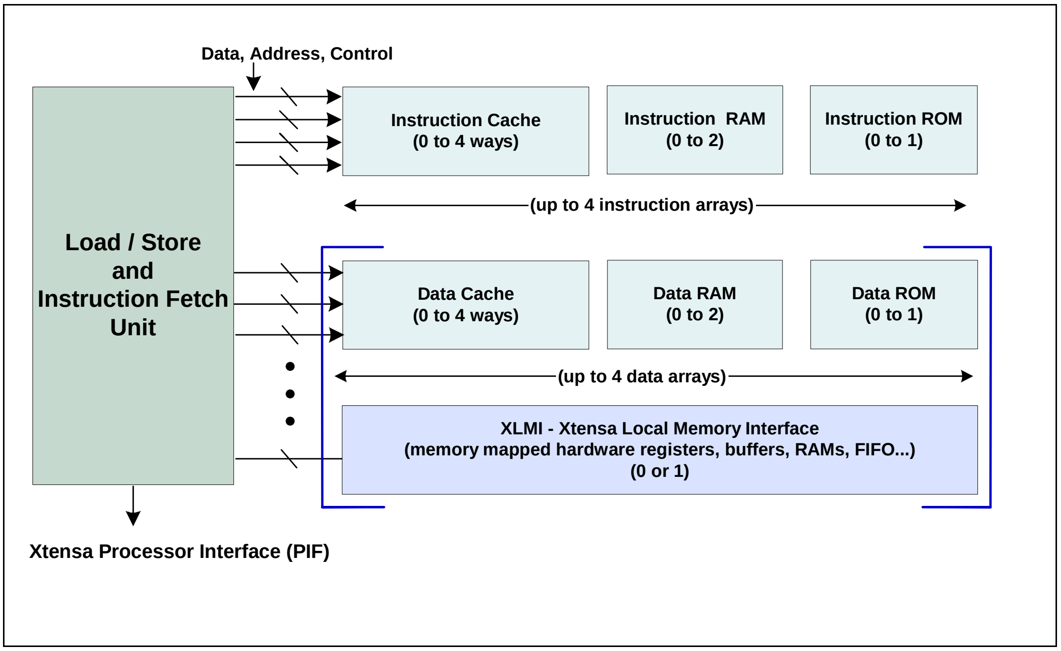

Data, Address, Control
Load / Store and Instruction Fetch Unit
Instruction Cache (0 to 4 ways)
Instruction RAM (0 to 2)
Instruction ROM (0 to 1)
(up to 4 instruction arrays)
Data Cache (0 to 4 ways)
Data RAM (0 to 2)
Data ROM (0 to 1)
(up to 4 data arrays)
XLMI - Xtensa Local Memory Interface (memory mapped hardware registers, buffers, RAMs, FIFO...) (0 or 1)
Xtensa Processor Interface (PIF)

Figure 33. Local Memory Options

9.3.1 Caches

The Xtensa LX processor can be configured to have separate data and instruction caches to significantly improve processor performance. Each cache can be configured to have different sizes (up to 32KB), different line sizes (16, 32 or 64 bytes), and different associativities (1-4 ways) depending on the application requirements. The data cache can also be configured to be write-through or write-back. A cache-locking option allows cache lines to be fetched and locked down in the cache to avoid cache misses in critical code.

Note: This option requires 2 or more ways of associativity.

The Xtensa ISA as implemented on the Xtensa LX processor ISA provides instructions that allow the application code to manage and test the caches. These instructions allow the code to invalidate cache lines, directly read and write the cache arrays, and lock cache lines in the cache.

Each cache way actually consists of two memory arrays: one array for the data and one for the cache tag. The Xtensa LX processor's data cache has a store-tag buffer with a 4-entry hardware structure that queues clean-to-dirty transitions to the data cache's tag array. This structure only exists in Xtensa LX processor configurations with the write-back cache option. The store tag buffer enhances the performance of dirty stores to the data caches.

9.3.2 RAM/ROM/XLMI Ports

The Xtensa LX processor can be configured to have dedicated memory ports for one or several local instruction memories (instruction RAM, instruction ROM), several data memories (data RAM, data ROM), and the special-purpose Xtensa Local Memory Interface (XLMI). Local Xtensa LX memories do not incur the same overhead as a cache way, because they lack tag arrays and data is not dynamically brought into and cast out of the local memories as it is with a cache. As many as two instruction-RAM and one instruction-ROM port can be configured. As many as two data-RAM ports, one data-ROM port, and one XLMI port can be configured. The base addresses for the local-memory and XLMI ports are configurable, and the address blocks reserved for these memories must not overlap.

Each RAM, ROM, and XLMI port has a configurable busy signal that can be used to indicate that the memory is being used by some logic external to the processor and that the memory or port is not available to the processor for a load or store transaction. This signal can be used to, among other things, share a single RAM or ROM between multiple processors, or to allow external logic to load the local memory.

Inbound-PIF operations to local RAM can also be configured to allow read and write transfers directly from the PIF to the local memories or the XLMI port.

9.4 Buses

The Xtensa LX Processor Interface (PIF) is an optional and configurable interface port for the Xtensa LX CPU core. It provides separate input and output data/address ports, configurable address and bus widths, configurable burst transfer sizes, and an inbound PIF request option. It allows single data transactions (less than or equal to the size of the data buses) as well as block transactions where several data bus widths of data are input or output using multiple bus cycles. It supports a split-transaction protocol with multiple outstanding requests. The PIF protocol does not dictate any arbitration protocol, and is easily adaptable to a variety of on-chip communication schemes.

10. Configurable Hardware Features

The Xtensa LX processor core provides designers with a very wide range of configuration options and extensions. These configurable features allow SOC developers to mold the abilities and performance of the processor cores used on the SOC to precisely fit the requirements of the various application tasks performed by the SOC. The Xtensa LX processor's configurable options include:

Core Microarchitecture Options

- 5- or 7-stage pipeline
  A 5-stage pipeline corresponds to a memory-access latency of one cycle. A 7-stage pipeline corresponds to a memory-access latency of two cycles. The default Xtensa LX pipeline has five stages: instruction fetch, register access, execute, data-memory access, and register writeback. This is a typical 5-stage RISC pipeline and it is sufficient for many uses on an SOC. However, as memories get larger and logic gets faster, slow local memories are becoming a growing problem for SOCs that use the most advanced IC processes.

  SOC designers using the Xtensa LX processor can compensate for those mismatches by using this configuration option to extend the default 5-stage pipeline to seven stages, which adds an extra instruction-fetch stage and an extra memory-access stage. The extra stages can effectively double or triple the available memory-access time at high clock frequencies.
- Number of registers
  32 or 64
- Number of Load/Store Units
  1 or 2
- Configurable Big- or Little-Endian Operation
- Unaligned Load/Store Action
  An unaligned load or store operation can either force an alignment by ignoring the lower bits of the address, or cause an exception.

10.1 Core Instruction Options

- Code Density
  The basic Xtensa LX instructions are encoded in either 16 or 24 bits for optimal code density. The 16- and 24-bit instructions can be freely intermixed in the instruction stream. The 16-bit instructions can be optionally deleted to save processor hardware at the expense of code density.

- Compound, Wide Instructions (FLIX)
  Compound 32- or 64-bit FLIX (Flexible Length Instruction Xtensions) instructions can be defined in TIE by the designer and added to the Xtensa LX instruction set. The 32- or 64-bit instructions can be freely intermixed with 24- and 16-bit instructions.
  Note: An Xtensa LX processor configuration can have 32- or 64-bit FLIX instructions, but not both.
- Zero Overhead Loop
  The zero-overhead-loop instructions and logic can be removed to save area.
  Note: Removing the zero-overhead-loop instructions may cause binary code incompatibility with certain commercial 3rd-party software such as real-time operating systems.
- MAC16
  This option adds a 16x16-bit multiplier and a 40-bit accumulator, eight 16-bit (or four 32-bit) operand registers, load instructions for the additional operand registers, and a set of compound instructions that combine multiply, accumulate, load, and automatically update operand addresses.
- MUL16
  This option adds a 16x16-bit multiplier that produces a 32-bit result.
- MUL32
  This option adds a 32x32-bit multiplier that produces a 64-bit result. The Mul32High option controls whether the upper 32-bits of the 64-bit result are made available to the software.
- EXTL32R
  This option increases the effective range of the L32R (Load 32-bit Relative) instructions by allowing the memory reference to be relative to a base special register instead of PC-relative.
- Synchronization
  This option implements load-acquire and store-release instructions that permit the management of load/store operation ordering.
- Conditional Store Synchronization
  This option implements the store conditional instruction, S32C1I, which is used for updating synchronization variables in memory shared by multiple interrupt handlers, multiple processes, or multiple processors.
- Miscellaneous Instructions
  These options implement miscellaneous instructions including Normalization Shift Amount instructions (NSAU, NSA), the Count One Bits instruction (POPC), the Cyclic Redundancy Check Instruction (CRC8), Minimum and Maximum Value instructions (MIN, MAX, MINU, MAXU), the Clamp instruction (CLAMP), and the Sign Extend instruction (SEXT).

10.2 Coprocessor Options

- Coprocessor
  This option adds state (processor registers) and instructions common to all coprocessors.
- IEEE754 Floating-Point Unit (FPU)
  This 32-bit single precision floating point coprocessor option is optimized for printing, graphics, and audio applications. An IEEE754 compatible FPU, this coprocessor provides the programming ease of floating-point arithmetic with the execution speed of fixed-point processing.
  Note: For best performance results, Xtensa LX configurations incorporating the FPU should be synthesized using Synopsys' DC Ultra.
- Vectra LX DSP Engine
  The Vectra LX DSP Engine is optimized to handle high-performance digital signal processing applications using fixed-point arithmetic. This SIMD (single-instruction, multiple-data) DSP option is ideal for communications, audio, and imaging applications employing a highly efficient and easy-to-program architecture.
  Note: Full best performance results, Xtensa LX configurations incorporating the Vectra LX DSP engine should be synthesized using Synopsys' DC Ultra.
- Designer-Defined Coprocessors
  Designers can define additional, application-specific coprocessors using the TIE language. Generic operations common to all coprocessors, such as moving data to and from designer-defined registers, or branch instructions that branch on the result of a coprocessor operation, are automatically provided by the generic Coprocessor Option.

10.3 Exception and Interrupt Options

The Xtensa LX processor supports the Xtensa Exception Architecture 2 (XEA2), which implements ring protection and adds flexibility for register-window handlers. The following exception options are available:

- Interrupt Option
  This option adds the ability to handle as many as 32 asynchronous exceptions generated from external and certain internal sources. The interrupt option on its own implements Level-1 interrupts, which are the lowest priority interrupts. Level-1 interrupts share one user- and one supervisor-level interrupt vector.
- Number of Interrupts
  Configurable, 1 to 32. The total number interrupts is equal to the number of Level-1 interrupts plus the number of high-priority, non-maskable, and timer interrupts. Note that interrupts caused by instruction exceptions are decoupled from the option that sets the number of available interrupts and are not a factor in the 32-interrupt limit.

- Interrupt Types
  - *External Level*—Level-sensitive input interrupt signal to the processor.
  - *External Edge*—Edge-triggered input interrupt signal to the processor.
  - *Internal*—An interrupt generated by internal processor logic (for example: timer and debug interrupts).
  - *Software*—An interrupt generated by software manipulation of read/write bits in the processor's `INTERRUPT` register.
  - *Non-maskable (NMI)*—External, edge-triggered interrupt input signal to the processor with an implicitly infinite priority level.
- High-Priority Interrupts and Non-Maskable Interrupt
  These options implement a configurable number of interrupt levels between levels 2 and 15, and an optional non-maskable interrupt (NMI). Each of the optional high-priority interrupts and the one optional non-maskable interrupt have individual interrupt vectors.
- C-callable Interrupts
  The `EXCMLEVEL` parameter, part of the high-priority interrupt option, allows the designer to designate the number of high-priority interrupts that are C-callable.

  Note: Level-1 interrupt handlers may always be written in C.
- Timer Interrupts
  This option implements from one to three internally-generated timer interrupts based on special registers that contain interrupt-trigger values. The processor compares the values in these registers with a 32-bit counter that is incremented each processor clock cycle. Interrupts occur when the values in the comparison and counter registers match.
- Write-Error Interrupt
  This option creates an interrupt source that generates an interrupt for a write-response error.

  Note: This option requires the PIF Write Response Option to be configured.
- Interrupt Vectors
  High-Priority and Non-Maskable Interrupts have individual interrupt vectors.

10.4 Memory Management Features

- Region Protection
  All Xtensa LX processors have an instruction TLB (ITLB) and a data TLB (DTLB) that divide the processor's address space into eight memory regions. These TLBs provide region protection for memory. The access modes of each memory region can be changed by writing to the appropriate TLB entries.

  Note: The Xtensa LX TLBs do not translate addresses (memory addresses are always identity mapped through the Xtensa LX TLBs).

10.5 Cache and Cache Port Options

The Xtensa LX processor core offers the following instruction- and data-cache options:

- Data Cache
  0, 1, 2, 4, 8, 16, and 32 Kbytes (direct-mapped, 2- or 4-way set-associative cache)
  0, 1.5, 3, 6, 12, and 24 Kbytes (3-way set-associative cache)
- Instruction Cache
  0, 1, 2, 4, 8, 16, 32 Kbytes (direct-mapped, 2- or 4-way set-associative cache)
  0, 1.5, 3, 6, 12, 24 Kbytes (for 3-way set-associative cache)
- Cache Line Size
  16, 32, or 64 bytes
- Cache Access Width
  The Xtensa LX processor’s PIF bus width can be configured to be 32, 64, or 128 bits wide. The access width of instruction and data cache memories matches the width of the PIF read interface. For example, if the PIF read width is set to 128-bits, the cache interface is also 128-bits.

  Note: Xtensa LX processors configured with caches must be configured with a PIF to service cache refill and castout operations.
- Cache-Memory Access Latency
  1 or 2 cycle access latency for 5-stage and 7-stage pipelines respectively.
- Associativity and Cache-Line Refill Method
  Direct-mapped or 2, 3, or 4-way set-associative cache
  Least-recently-filled (LRF) per cache-line index refill algorithm
- Data-Cache Write Policies
  Write-Through
  Write-Back
- Cache-Line Locking (for 2, 3, or 4-way set-associative caches)
- Cache Test Instructions
  Instructions to read and write data and tag arrays directly for testing purposes

10.6 RAM Port Options

The Xtensa LX processor core offers the following RAM port options:

- 0, 1, or 2 Data RAMs
  sizes of 0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes
- 0, 1, or 2 Instruction RAMs
  sizes of 0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes

Note: Instruction RAM sizes beyond 128 Kbytes require the `ExtL32R` option.

- RAM Access Width
  The RAM ports can be configured to be 32, 64, or 128 bits wide. The access width of the data RAM interface matches the processor’s read/write data-bus width. The access width of the instruction RAM interface is fixed at either 32 bits (if there are no 64-bit wide instructions) or 64 bits (if there are 64-bit instructions). For No-PIF processor configurations, the data-access width is directly specified instead of being inferred from the configured PIF width.
- RAM Port Latency
  1- or 2-cycle access latency for 5-stage and 7-stage pipelines respectively.
- RAM Busy
  Data- and instruction-RAM ports have optional busy inputs that allow these local memories to be shared among multiple processors or initialized externally.
- Inbound-PIF Operations to RAM
  Both data- and instruction-RAM ports optionally permit inbound-PIF read and write operations to the attached local RAMs.
- Base Physical Address
  The starting physical address for each RAM is configurable. The starting address must be aligned to the RAM size.

10.7 ROM Port Options

The Xtensa LX processor core offers the following ROM port options:

- 0 or 1 Data ROM
  0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes
- 0 or 1 Instruction ROM
  0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes

  Note: Instruction ROM sizes beyond 128 Kbytes require the `ExtL32R` option.
- ROM Port Access Width
  The ROM ports can be configured to be 32, 64, or 128 bits wide. The access width of the Data ROM interface matches the processor read/write data bus width. The access width of the Instruction ROM interface is fixed at either 32 bits (if there are no 64-bit instructions) or 64 bits (if there are 64-bit wide instructions). For No-PIF processor configurations, the data access width is directly specified in the configuration generator instead of being inferred from the configured PIF width.
- ROM Latency
  1- or 2-cycle access latency for 5-stage and 7-stage pipelines respectively.
- ROM Busy
  Both data- and instruction-ROM ports have optional busy inputs that allow these local memories to be shared among multiple processors or initialized externally.

- Base Physical Address
  The starting physical address for each ROM is configurable. The starting address must be aligned to the ROM size.

10.8 Xtensa Local Memory Interface (XLMI) Port Options

The Xtensa LX processor core offers the following XLMI port options:

- One optional XLMI port, mapped to an address space of size:
  0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes.
- XLMI Port Access Width
  The access width of the XLMI port matches that of the PIF and can be 32, 64, or 128 bits wide. For No-PIF processor configurations, the XLMI port width is directly specified in the configuration generator instead of being inferred from the configured PIF width.
- XLMI Port Latency
  1- or 2-cycle access latency for 5-stage and 7-stage pipelines respectively.
- XLMI Busy
  The XLMI port has an optional busy input that allows -9(o)-0.3(wse9(nput8 8 i)-9((s)3.4(6Q

The Xtensa LX processor generator uses the System ROM and RAM options to set up a default memory map, which affects the software-linker and hardware-diagnostic scripts automatically created by the generator. Defining the size and base addresses for the system memories also creates a default memory map that can run out-of-the-box C applications.

10.10 Processor Interface (PIF) Options

- No-PIF configuration
  The designer can eliminate the processor interface (PIF) using a configuration option to reduce gate count. A processor configured without a PIF must operate entirely out of local memory.

  Note: Because Xtensa LX processor cache-line fills and castouts will use the PIF to communicate with system memory, a processor configured without a PIF cannot support and therefore may not have cache memories.
- PIF Width Option
  The PIF can be configured to be 32, 64, or 128 bits wide. The PIF read and write data buses are the same width.
- Inbound-PIF Request Option
  Supports inbound-PIF requests to instruction RAM, data RAM, and the XLMI port so that external logic including other processors and DMA controllers can access the Xtensa LX processor's local-memory ports via the PIF.
- Inbound-PIF-Request Buffer Size
  Sets the size of the inbound-PIF request buffer. The buffer size must be equal to or larger than the largest inbound-PIF block request size.
- Write-Buffer Entry Capacity
  The write buffer size can be configured to hold 4, 8, 16, or 32 entries.
- Write Response Option
  The Xtensa LX processor can be configured to count write responses for memory ordering or to aid in the use of synchronization primitives. When write responses are configured, 16 unique IDs are allocated to write requests. As many as 16 write requests can be outstanding at any time.
- Inbound-PIF Write-Response Option
  This configuration option causes the PIF to send write responses to external devices that issue inbound-PIF write requests.

Note: If the inbound-PIF option is configured, the Xtensa LX processor will return write responses for incoming write requests.

10.11 TIE Module Options

- TIE Arbitrary-Byte-Enable Option
  Allows TIE-based load/store instructions to initiate bus operations with arbitrary byte-enable patterns.
- TIE Use-Wide-Store Option
  Allows TIE-based store instructions to use the full PIF width for store operations. This option affects the write-buffer width, which can be 32, 64, or 128 bits. Without this option, TIE-based loads can still use the full PIF width for transactions, but stores are restricted to 32-bit transactions.

10.12 Test and Debug Options

- Scan Option
  Implements scan chains for SOC testability.
- Debug Option
  Implements instruction counting and breakpoint exceptions for debugging software or for other external-hardware uses. This option requires the high-priority interrupt option, requires the setting of a debug interrupt level, and requires the selection of the number of instruction breakpoint registers (0, 1, or 2) and the number of data breakpoint registers (0, 1, or 2).
- On-Chip Debug (OCD) Option
  Enables hardware access to software-visible processor state through an IEEE 1149.1 test-access port (TAP).
- Trace Port Option
  Creates status outputs that reflect the processor’s internal state during normal operation. The option’s data-trace sub-option parameter allows the trace port to report additional information about register and load/store values.

10.13 Power Management Options and Features

- WAITI Option
  Allows the processor to suspend operation until an interrupt occurs by executing the optional WAITI instruction.

  Note: This option is part of the interrupt option.
- Global Clock-Gating Option
  Allows the processor generator to add clock gating to the main processor clock, which is routed to large portions of the processor’s internal logic. This option can be used in conjunction with the functional clock-gating option.

- Functional Clock-Gating Option
  Allows the processor generator to add clock gating to the processor's registers so that register clocking is based on register usage. This option can be used in conjunction with the global clock-gating option.
- External Run/Stall Control Signal
  This processor input allows external logic to stall large portions of the processor's pipeline by shutting off the clock to much of the processor's logic to reduce operating power when the Xtensa LX processor's computational capabilities are not immediately needed by the system.

  Note: Using the `WAITI` instruction to power down the Xtensa LX processor will save more power than use of the external run/stall signal because the `WAITI` instruction will disable more of the internal processor clocks.

10.14 Additional Configuration Options

- Miscellaneous Special Register Option
  As many as four optional scratch registers can be configured. Software accesses these registers using the RSR, WSR, and XSR special-register instructions.
- Processor ID option
  Creates a processor ID (PRID) register that latches the state of the 16-bit PRID processor input bus during reset. The PRID register option is available to Xtensa LX licensees who have purchased the System Modeling and Multiprocessor Support package.

11. Bus and Signal Descriptions

Figure 34 shows all of the configurable ports going into and out of Xtensa LX. Which ports are actually present, as well as the width of many of the buses, depends on the specific configuration. Short descriptions of these ports appear in Tables 71 through 85 in the sections that follow.

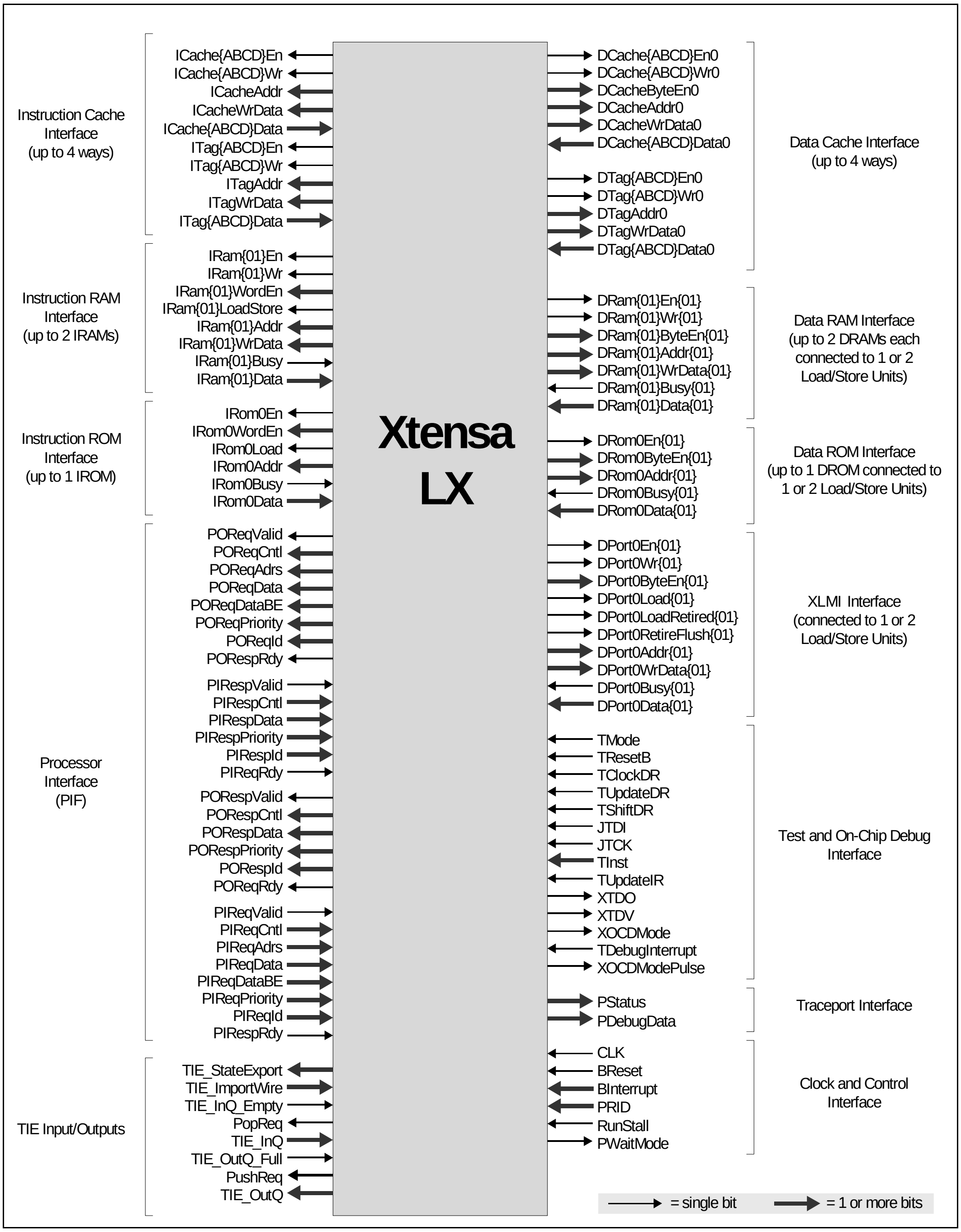


Figure 34. Xtensa LX Block Diagram Showing All Possible Ports

11.1 Xtensa LX Signal Names and Short Descriptions

Table 71. Clock and Control Interface Signals

Name	Direction	Config Option	Description
`CLK`	Input	Any	Clock input to the processor (50% duty cycle required). Note: `CLK` is used to generate the internal processor clock signal `PCLK`.
`BReset`	Input	Any	Reset input to the processor. This signal may be asserted asynchronously, but must be negated synchronously with `PCLK`. The reset signal is registered for two cycles before it is used internally. `BReset` is active high. A minimum of 50 `PCLKs` are required to reset the Xtensa LX processor.
`PWaitMode`	Output	Interrupt	Indicates that the processor is in sleep mode. The processor asserts this signal when it has executed a `WAITI` instruction and is waiting for an interrupt. Any asserted interrupt that is not disabled will wake the processor, which will then jump to the appropriate designer-configured interrupt vector.
`RunStall`	Input	Any	This external signal stalls the processor. This signal is registered within the Xtensa LX processor before it is used.
`PRID[15:0]`	Input	PRID	Input to the processor, latched at reset into the low-order bits of the PRID (processor ID) special register. The PRID register option is available to Xtensa LX licensees who have purchased the System Modeling and Multiprocessor Support package.
`BInterrupt` [number of interrupts-1:0]	input	Interrupt	External processor interrupts. As many as 32 interrupt inputs can be configured. Each interrupt input can be configured to be level- or edge-triggered. If the processor is not busy, assertion of an interrupt input causes code execution to jump to the appropriate interrupt handler routine at a specified vector location. One of the interrupt bits can be configured by the designer to be a non-maskable interrupt (NMI). Note: Certain conditions can delay interrupts.

Table 72. Test and On-Chip Debug Interface Signals

Name (configured width)	Direction	Config Option	Description
`JTCK`	Input	OCD	Standard 1149.1 JTAG TAP clock.
`JTDI`	Input	OCD	Standard 1149.1 JTAG TAP port serial data in.
`TResetB`	Input	OCD	Reset signal from TAP Controller. (Asserted LOW).
`TShiftDR`	Input	OCD	TAP state machine in Shift-DR state.
`TClockDR`	Input	OCD	Standard 1149.1 JTAG TCK when the TAP state machine in Capture-DR or Shift-DR state.

Table 72. Test and On-Chip Debug Interface Signals (continued)

Name (configured width)	Direction	Config Option	Description
`TUpdateDR`	Input	OCD	TAP state machine in Update-DR state.
`TUpdateIR`	Input	OCD	TAP state machine in Update-IR state.
`TInst[4:0]`	Input	OCD	Instruction inputs into the TAP Instruction Register.
`XTDO`	Output	OCD	Test Data Output from the processor's JTAG Data Registers.
`XTDV`	Output	OCD	Indicates that the Test Data Output (XTDO) from the processor is valid.
`XOCDMode`	Output	OCD	Indicates that the processor is in OCD halt mode.
`XOCDModePulse`	Output	OCD	A 1-cycle pulse from the processor indicating that it has entered OCD halt mode.
`TDebugInterrupt`	Input	OCD	External debug interrupt.
`PStatus` [pstatuswidth-1:0][1]	Output	Traceport	Processor debug status during W stage.
`PDebugData` [pdebugwidth-1:0][2]	Output	Traceport	Processor debug data during W stage.
`TMode`	Input	Scan	Special test mode signal for scan testing.

Notes:

1. pstatuswidth = 8 if only address trace is configured and 16 if data trace is configured

2. pdebugwidth = 32 if only address trace is configured and 64 if data trace is configured

Table 73. Instruction RAM Interface Port Signals

Name (configured width)	Direction	Config Options	Description
`IRam`*n*`Addr [irambyteaddrwidth-1:isize]`[1,2,3]	output		Instruction RAM address lines.
`IRam`*n*`Busy`[1]	input	IRamBusy	Instruction RAM memory unavailable during the previous cycle. Note: There are combinational paths from `IRam`*n*`Busy` to all memory address, data, and control outputs.
`IRam`*n*`Data [iramwidth-1:0]`[1,4]	input		Input data lines from instruction RAM.
`IRam`*n*`En`[1]	output		Instruction RAM enable.

Table 73. Instruction RAM Interface Port Signals (continued)

Name (configured width)	Direction	Config Options	Description
`IRam`*n*`WordEn [1:0]`[1]	output	64-bit-wide instructions	Word enables for 64-bit-wide instruction RAMs. Instruction RAM loads and stores are always 32 bits.
`IRam`*n*`LoadStore`[1]	output		Load or store is occurring from/to instruction RAM.
`IRam`*n*`Wr`[1]	output		Instruction RAM write line.
`IRam`*n*`WrData [iramwidth-1:0]`[1,4]	output		Data to be written to instruction RAM.

Notes:

1. n is the number of the associated instruction RAM (*n* = 0 if there is one RAM and *n* = {0, 1} if there are two RAMs.)

2. isize = 3 if 64-bit FLIX instructions are configured, 2 otherwise

3. irambyteaddrwidth = log2(IRam size [in bytes])

4. iramwidth = IRam access width [in bits]

Table 74. Data RAM Interface Port Signals

Name (configured width)	Direction	Config Option	Description
`DRam`*n*`Addr`*m* `[drambyteaddrwidth-1:dramaccessoffset]`[1,2,3,6]	output		Data RAM address lines.
`DRam`*n*`ByteEn`*m* `[drambytes-1:0]`[1,2,5]	output		Data RAM byte enables.
`DRam`*n*`Data`*m* `[dramwidth-1:0]`[1,2,4]	input		Input data lines from data RAM.
`DRam`*n*`En`*m*[1,2]	output		Data RAM enable.

Notes:

1. n is the number of the associated data RAM (*n*=0 if the configuration has one data RAM. *n*={0, 1} if the configuration has two data RAMs.)

2. m is the number of the associated load/store unit

3. drambyteaddrwidth = $\log_2$(DRam size [in bytes])

4. dramwidth = DRam access width [in bits]

5. drambytes = dramwidth / 8

6. dramaccessoffset = $\log_2$(drambytes)

Table 74. Data RAM Interface Port Signals (continued)

Name (configured width)	Direction	Config Option	Description
`DRam`*n*`Busy`*m*[1,2]	input	DRamBusy	Data RAM memory unavailable during the previous cycle. Note: There are combinational paths from `DRam`*n*`Busy` to all memory address, data, and control outputs.
`DRam`*n*`Wr`*m*[1,2]	output		Data RAM write line.
`DRam`*n*`WrData`*m* `[dramwidth-1:0]`[1,2,4]	output		Data to be written to data RAM.

Notes:

1. *n* is the number of the associated data RAM (*n*=0 if the configuration has one data RAM. *n*={0, 1} if the configuration has two data RAMs.)

2. *m* is the number of the associated load/store unit

3. drambyteaddrwidth = $\log_2$(DRam size [in bytes])

4. dramwidth = DRam access width [in bits]

5. drambytes = dramwidth / 8

6. dramaccessoffset = $\log_2$(drambytes)

Table 75. Instruction ROM Interface Port Signals

Name (configured width)	Direction	Config Options	Description
`IRom0Addr [irombyteaddrwidth-1:isize]`[1,2]	output		Instruction ROM address lines.
`IRom0Busy`	input	IRomBusy	Instruction ROM memory unavailable during the previous cycle. Note: There are combinational paths from `IRom`*n*`Busy` to all memory address, data, and control outputs.
`IRom0Data[iromwidth-1:0]`[3]	input		Input data lines from instruction ROM.
`IRom0En`	output		Instruction ROM enable.
`IRom0WordEn[1:0]`	output	64-bit wide instructions	Word enables for 64-bit-wide instruction ROMs. Instruction ROM loads are always 32 bits.
`IRom0Load`	output		Load is occurring from instruction ROM.

Notes:

1. isize = 3 if 64-bit FLIX instructions are configured, 2 otherwise

2. irombyteaddrwidth = log2(IRom size [in bytes])

3. iromwidth = IRom access width [in bits]

Table 76. Data ROM Interface Port Signals

Name (configured width)	Direction	Config Options	Description
`DRom0Addr`*m* `[drombyteaddrwidth-1:dromaccessoffset]`[1,2,4]	output		Data ROM address lines.
`DRom0Data`*m* `[dromwidth-1:0]`[1,3]	input		Input data from data ROM.
`DRom0En`*m*[1]	output		Data ROM enable.
`DRom0Busy`*m*[1]	input	DRomBusy	Data ROM memory unavailable during previous cycle. Note: There are combinational paths from `DRom`*n*`Busy` to all memory address, data, and control outputs.

Note:

1. *m* is the number of the associated load/store unit
2. drombyteaddrwidth = log2(DRom size [in bytes])
3. dromwidth = DRom access width [in bits]
4. dromaccessoffset = log2(dromwidth / 8)

Table 77. Instruction Cache Interface Port Signals

Name (configured width)	Direction	Config Option	Description
`ICache<`*way*`>Data` `[icwidth-1:0]`[5]	input		Input data from instruction cache array (per way).
`ICache<`*way*`>En`	output		Instruction cache array enable (per way).
`ICache<`*way*`>Wr`	output		Instruction cache write enable (per way) — valid during refills and special cache instructions.
`ICacheAddr` `[icsetawidth+icwordoffset-1: icwordoffset]`[2,3]	output		Instruction cache array address lines.
`ICacheWrData` `[icwidth-1:0]`[5]	output		Output data to be written to instruction cache array.
`ITag<`*way*`>Data` `[itwidth-1:0]`[4]	input		Data read from instruction cache tag array (per way).
`ITag<`*way*`>En`	output		Instruction cache tag array enable (per way).

Table 77. Instruction Cache Interface Port Signals (continued)

Name (configured width)	Direction	Config Option	Description
`ITag<way>Wr`	output		Instruction cache tag array write enable (per way).
`ITagAddr [icsetbyteaddrwidth-1: iclineoffset]`[1,8]	output		Instruction cache tag array address lines.
`ITagWrData [itwidth-1:0]`[4]	output		Output data to be written to instruction-cache tag array.

Notes:

1. icsetbyteaddrwidth= log2(ICache size [in bytes]/ways)
2. icsetawidth = log2((Icache size/4)/ways)
3. icwordoffset = log2(icwidth/8)
4. itwidth = ITag access width [in bits]
5. icwidth = ICache access width [in bits]
6. icbytes = icwidth / 8
7. icaccessoffset = log2(icbytes)
8. iclineoffset = log2(ICache line size [in bytes])

Table 78. Data Cache Interface Port Signals

Name (configured width)	Direction	Config Option	Description
`DCache<way>Data [dcwidth-1:0]`[3]	input		Input data from data cache array (per way).
`DCache<way>En`	output		Data cache array enable (per way).
`DCache<way>Wr`	output		Data cache write-enable (per way).
`DCacheAddr [dcsetbyteaddrwidth-1: dcaccessoffset]`[1,5]	output		Data cache array address lines.
`DCacheByteEn [dcbytes-1:0]`[4]	output		Data cache byte enables.
`DCacheWrData [dcwidth-1:0]`[3]	output		Output data to be written to data cache array.
`DTag<way>Data [dtwidth-1:0]`[2]	input		Input data read from data cache tag array (per way).
`DTag<way>En`	output		Data cache tag array enable (per way).
`DTag<way>Wr`	output		Data cache tag array write enable (per way).
`DTagAddr [dcsetbyteaddrwidth-1: dclineoffset]`[1,6]	output		Data cache tag array address lines.
`DTagWrData [dtwidth-1:0]`[2]	output		Data to be written to data cache tag array.

Notes:

1. dcsetbyteaddrwidth= $\log_2$(DCache size [in bytes]/ways)
2. dtwidth = DTag access width [in bits]
3. dcwidth = DCache access width [in bits]
4. dcbytes = dcwidth / 8
5. dcaccessoffset = $\log_2$(dcbytes)
6. dclineoffset = $\log_2$(DCache line size[in bytes])

Table 79. PIF Master Signals

Name (configured width)	Direction	Description
`POReqValid`	Output	Indicates that there is a valid bus-transaction output request. All other signals prefixed with `POReq` are qualified by `POReqValid`.
`POReqCntl[7:0]`	Output	Encodes bus-transaction request type and block size for block requests.
`POReqAdrs[31:0]`	Output	Transaction request address. Address is aligned to the transfer size.
`POReqData [pifwidth-1:0]`	Output	Data lines used by requests that require data during the request phase. Requests include single data write, block write, and read-conditional-write requests. The data bus is configurable to 32, 64 or 128-bits, to provide the target bandwidth required by a system or application.
`POReqDataBE [pifwidth/8-1:0]`	Output	Byte enables indicating valid lanes during requests that use `POReqData`, or byte lanes expected to be valid during responses that use `PIRespData`.
`POReqId[5:0]`	Output	Request ID. Responses are expected to be tagged with the same ID as the request on `PIRespId`. IDs support multiple outstanding load and store requests. No restrictions or orderings are placed on the usage or uniqueness of IDs.
`POReqPriority[1:0]`	Output	Request priority allows PIF slaves and interconnection networks to make priority judgments when arbitrating bus-transaction requests. 0x0 is low-priority, and 0x3 is high-priority.
`PIReqRdy`	Input	Indicates that the PIF slave is ready to accept requests. A request transfer completes if `PIReqRdy` and `POReqValid` are both asserted during the same cycle.
`PIRespValid`	Input	Indicates that there is a valid response. All other signals prefixed with `PIResp` are qualified by `PIRespValid`.
`PIRespCntl[7:0]`	Input	Encodes response type and any error status for transaction requests.
`PIRespData [pifwidth-1:0]`	Input	Response data. The data bus is configurable to 32, 64, or 128-bits and is equal in width to the request data bus.
`PIRespId[5:0]`	Input	Response ID. Matches the ID of the corresponding request.
`PIRespPriority[1:0]`	Input	Response priority. Matches the priority of the corresponding request.
`POResp Rdy`	Output	Indicates that the PIF master is ready to accept responses. A response transfer completes when `PIRespValid` and `POResp Rdy` are asserted during the same cycle.

Note:

1. pifwdith = configured PIF width, one of 32, 64, or 128-bits.

Table 80. PIF Slave Signals

Name (configured width)	Direction	Description
`PIReqValid`	Input	Indicates that there is a valid bus-transaction input request. All other signals prefixed with `PIReq` are qualified by `PIReqValid`.
`PIReqCntl[7:0]`	Input	Encodes the bus-transaction request type and block size for block requests.
`PIReqAdrs[31:0]`	Input	Transaction-request address. Address is aligned to the transfer size.
`PIReqData [pifwidth-1:0]`	Input	Data lines used by requests that require data during the request phase. These requests include single data write, block write, and read-conditional-write requests. The data bus is configurable to 32, 64, or 128-bits.
`PIReqDataBE [pifwidth/8-1:0]`	Input	Indicates valid bytes lanes during requests that use `PIReqData`, or byte lanes expected to be valid during responses that use `PORespData`.
`PIReqId[5:0]`	Input	Request ID. Responses are expected to be tagged with the same ID as the request on `PORespId`. IDs support multiple outstanding requests. No restrictions or orderings are placed on the usage or uniqueness of IDs.
`PIReqPriority[1:0]`	Input	Request priority allows PIF slaves and interconnection networks to make priority judgments when arbitrating bus-transaction requests. 0x0 is low-priority, and 0x3 is high-priority.
`POReqRdy`	Output	Indicates that the slave is ready to accept requests. A request transfer completes if `POReqRdy` and `PIReqValid` are both asserted during the same cycle.
`PORespValid`	Output	Indicates that there is a valid response. All other signals prefixed with `POResp` are qualified by `PORespValid`.
`PORespCntl[7:0]`	Output	Encodes the response type and any error status on requests.
`PORespData [pifwidth-1:0]`	Output	Response data. The data bus is configurable to 32, 64, or 128-bits and is equal in width to the request data bus.
`PORespId[5:0]`	Output	Response ID. Matches the ID of the corresponding request.
`PORespPriority[1:0]`	Output	Response priority. Matches the priority of the corresponding request.
`PIRespRdy`	Input	Indicates that the master is ready to accept responses. A response transfer completes when `PORespValid` and `PIRespRdy` are asserted during the same cycle.

Note:

1. pifwdith = configured PIF width, one of 32, 64, or 128-bits.

Table 81. XLMI (Xtensa Local Memory Interface) Port Signals

Name (configuration width)	Direction	Config Option	Assertion Timing[7, 8, 9]	Description
DPort0En*m*[2]	output		N_{En}	XLMI port *n* enable.
DPort0Addr*m* *[awidth-1:0]* [3]	output		N_{En}	Virtual address to XLMI port *n* - only bits [awidth-1:dportaccessoffset] are used
DPort0ByteEn*m* *[dportbytes-1:0]* [2, 5]	output		N_{En}	XLMI port *n* byte enables.
DPort0Wr*m*[2]	output		N_{En}	XLMI port *n* write enable.
DPort0WrData*m* [dportwidth-1:0] [2,4]	output		N_{En}	Data to be written to XLMI port *n.*
DPort0Busy*m*[2]	input	DPortBusy	$N_{En} + 1$	Load or Store transaction not accepted by XLMI port *n* (present only if Busy option selected). **Note:** There are combinational paths from DPort0Busy*m* to all memory address, data, and control outputs.
DPort0Load*m*[2]	output		$N_{En} + 1$	Load transaction to XLMI port *n* has begun.
DPort0Data*m* [dportwidth-1:0] [2, 4]	input		$N_{En} + N_{Access}$	Input data from XLMI port *n.*
DPort0LoadRetired*m*[2]	output		$N_{En} + N_{Access} + N_{Stall}$	Oldest outstanding load from XLMI port *n* committed.
DPort0RetireFlush*m*[2]	output		$N_{En} + N_{Access} + N_{Stall}$	All outstanding loads from XLMI port *n* will not commit.

Notes:

1. The assertion-timing column describes the cycle-time relationship between the signals. N_{en} is the cycle at which DPort*n*En*m* is asserted. N_{en}+1 for DPort*n*Busy*m* means that the busy signal is always expected one cycle after the assertion of the enable.

2. m = number of the associated load/store unit (0 or 1).

3. awidth = virtual address width

4. dportwidth = data access width of DPort*n*

5. dportbytes = dportwidth/8

6. dportaccessoffset = $\log_2$(dportbytes)

7. N_{En} = cycle that DPort*n*En*m* is asserted

8. N_{Access} = memory access latency i.e. 1 for a 5-stage pipeline or 2 for a 7-stage pipeline.

9. N_{Stal}l = arbitrary number of stall cycles

Table 82. TIE Input Queue Signals (per Input Queue instantiation)

Name (configured width)	Direction	Description
`TIE_<name> [iqwidth-1:0]`	Input	Input data bus (or wire if 1-bit width) connected to the output of the external queue. The data is registered in the Xtensa LX processor before it is used by any instruction semantics.
`TIE_<name>_PopReq`	Output	Indicates that the Xtensa LX processor is ready to read data from the input queue. This request is made independently of the Empty signal. The attached queue must ignore this signal if the queue is empty.
`TIE_<name>_Empty`	Input	Indicates the external queue is empty and cannot be popped.
Note: There is one set of these three interfaces per input queue instantiation. TIE code: queue <name> <iqwifth> in		

Table 83. TIE Input Signals (Import Wire, per Wire)

Name (configured width)	Direction	Description
`TIE_<name> [wirewidth-1:0]`	Input	A data value that is available for use in any TIE instruction.
Note: There is one interface per imported wire. TIE code: import_wire <name> <wirewidth>		

Table 84. TIE Output Queue Signals (per Output Queue instantiation)

Name (configured width)	Direction	Description
`TIE_<name> [oqwidth-1:0]`	Output	Output data bus (or wire if 1-bit width) connected to the input of the external queue. The data is registered in the Xtensa LX processor before it is sent to the external queue.
`TIE_<name>_PushReq`	Output	Indicates that the Xtensa LX processor is ready to write data to the output queue. This request is made independent of the status of the full signal and the attached queue is expected to ignore this signal if the queue is full.
`TIE_<name>_Full`	Input	Indicates the external queue is full and cannot be written to.
Note: There is one set of these three interfaces per output queue instantiation. TIE code: queue <name> <oqwifth> out		

Table 85. TIE Output (Export State, per Exported State)

Name (configured width)	Direction	Description
`TIE_<name>` `[statewidth-1:0]`	Output	Exported state data bus (or wire if 1-bit width). The architectural copy of the state is exported on these wires.
Note: There is one interface per state that is exported TIE code: state <name> <statewidth> export		

12. Processor Interface (PIF)

The Processor Interface (PIF) is a configurable, multi-master interface for the Xtensa LX microprocessor core. It is a versatile, high performance interface with the flexible features necessary to connect the Xtensa LX processor through a system-specific external agent or bus interface module to a wide variety of standard or proprietary buses.

The Xtensa LX processor can be configured to have a master PIF or a master and a slave PIF. A master PIF is used for *outbound* requests, and a slave PIF is used for *inbound* requests. All instruction and data cache misses and castouts are serviced through the outbound PIF, in addition to uncached and cache-bypass requests. An Xtensa LX processor can be configured with an inbound, slave PIF (inbound-PIF Request option) to handle inbound transaction requests to its local data-memory ports (the instruction-RAM, data-RAM, and XLMI ports).

12.1 PIF-Based Systems

Figure 35 illustrates two different ways that a system can be organized to connect Xtensa LX processors to memories, peripherals, and other processors through the PIF.

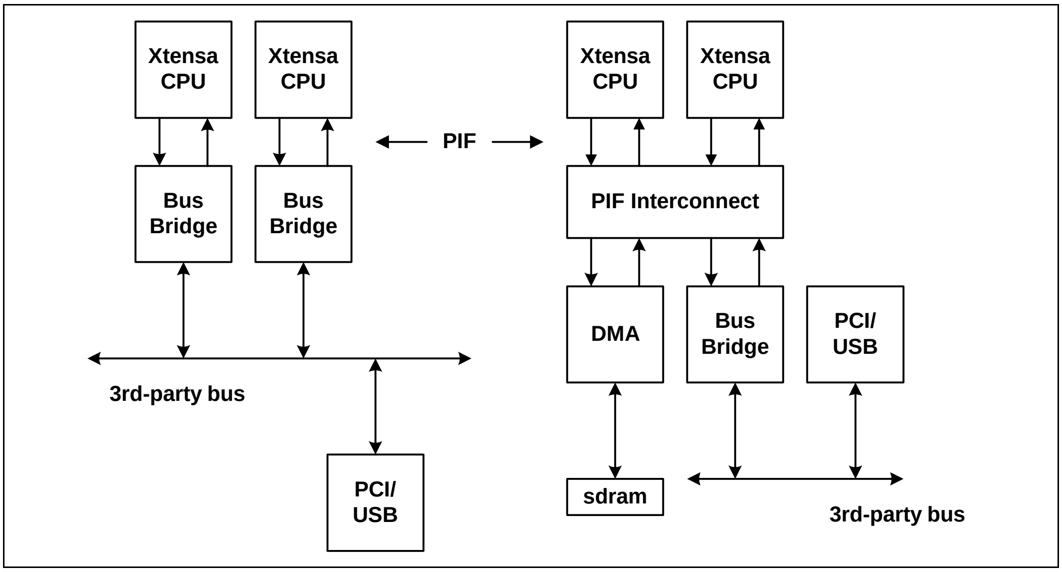


Figure 35. Examples of System Bus Interfaces Linked to Xtensa LX Processor Interface

12.2 Xtensa LX Processor Interface Features

This section gives an overview of the Xtensa LX PIF features. Other components incorporating PIFs may have similar features.

12.2.1 Synchronous and Pipelined

`CLK` is the input to the Xtensa LX processor core. `PCLK`, an internal Xtensa LX signal derived from `CLK`, represents a typical leaf node of the non-gated clock distribution tree within the core.

Input signals are sampled on the rising edge of `PCLK`, with some input delay and setup time requirement. Output signals are driven directly from registers on the rising edge of `PCLK`, with only clock-to-Q and routing output delay. This clocking scheme imposes a 1-cycle pipeline delay between every acknowledgement and response.

12.2.2 Unidirectional PIF Signals

All PIF signals are unidirectional.

12.2.3 Single Data Transactions

Read and write transactions smaller than or equal to the data-bus width require only one data-transfer cycle and are called single data transactions. Byte enables indicate which bytes should be read or written. The PIF can be configured with an optional write-response handshake mechanism to aid in reporting of transaction errors and for multi-processor synchronization. The Xtensa LX processor issues single data transaction requests for uncached and cache-bypass requests, and for cache misses when a cache line fits within the configured PIF data-bus width.

12.2.4 Block Transactions

Block-read and -write transactions allow block data transfers that use multiple full-width PIF transaction cycles. The Xtensa LX processor issues block requests for cache misses and castouts when the cache line is larger than the configured PIF data-bus width. Block sizes of 2, 4, 8, and 16 transfers are allowed. The PIF can be configured with an optional write-response handshake mechanism to aid in reporting of transaction errors and for multi-processor synchronization.

12.2.5 Read Conditional Write Transaction

Read-conditional-write transactions perform an atomic update of a memory location and are used to implement synchronization primitives for inter-processor or inter-process communication by providing mutually-exclusive access to data (for example, locks).

Note: These operations only occur if the Xtensa LX synchronization option is configured.

12.2.6 Multiple Outstanding Requests

Multiple transaction requests may be outstanding on the PIF. Transaction responses may be returned out of order and dynamic flow-control mechanisms are available to control the number of requests that are outstanding at any time. Six bits of transaction ID are available for each transaction request and these bits can be used by the requester in any fashion without restriction.

Note: A system need not handle multiple transactions. The number of outstanding transactions can be kept to one by not asserting the PIF ready lines.

12.2.7 Flexible Arbitration for System Bus

The processor interface protocol does not dictate a specific arbitration protocol and is easily adaptable to a variety of schemes through an external agent. Priority bits are defined for every transaction and can be used for arbitrating transaction requests from multiple sources.

12.2.8 Full Handshake-Based Protocol

A PIF master begins a transaction by issuing a valid *request* assertion, and a PIF slave indicates that it is ready to accept requests by asserting a *request ready* signal. A request transfer completes when valid request and request-ready signals are both asserted during the same cycle. The flow of each transfer within a block transaction can be controlled with this handshake mechanism.

Similarly, a PIF slave completes a transaction by issuing a valid *response* to the PIF master's request. The master accepts the response by asserting a *response ready* signal. A response transfer completes when valid response and response-ready signals are both asserted during the same cycle. The flow of each transfer within a block response can be controlled with this handshake mechanism.

These per-transfer handshake mechanisms permit more flexible component design and arbitration schemes. A PIF master can throttle its outbound request stream and its incoming response stream to match data bandwidth to available buffering. Similarly, a PIF slave can throttle its outbound response stream and its incoming request stream to match data bandwidth to available buffering.

12.2.9 Narrow-Width Peripheral/Memory Support

There is no special support for I/O transactions to devices narrower than the PIF data buses. Xtensa LX cores will align loads and stores within the appropriate data bus as described in Section 13.7 “PIF Addresses and Byte Enables” on page 177.

Consequently, a byte-wide peripheral device requires an 8:1 mux to receive its data from any byte lane of 64-bit PIF data bus during a write transaction, and the peripheral device must drive its byte-wide data to the specified byte lane during a PIF read transaction.

A bus interface or external bus bridge, aware of the external peripheral/device requirements (endianness and data width), is needed to align data between the PIF and an external bus.

12.2.10 Write Buffer

The Xtensa LX processor implements a configurable-depth write buffer to queue store requests and cache-line castouts. The depth of the write buffer can be tuned to the PIF bandwidth available to a configured processor during the development process.

12.2.11 Inbound-PIF Option

A slave PIF may be configured to handle inbound requests to the Xtensa LX processor local memories, which include instruction and data RAMs and the XLMI port.

12.2.12 No-PIF Configuration

An Xtensa LX processor can be configured without a PIF for SOC designs that require the lowest possible gate count or for deeply embedded uses that do not require access to a global memory structure. Xtensa LX processors configured without a PIF must operate entirely from local memory and do not support cache memory.

12.3 PIF Configuration Options

The following configuration options are available:

- No-PIF option:
  The Xtensa LX processor interface is optional.

  Note: Xtensa LX processors configured with caches must be configured with a PIF to service cache refill and castout operations.
- Processor Data Bus Width:
  All request and response data buses in a PIF must be the same width and can be 32, 64, or 128 bits wide.
- Cache Block-Transfer Size:
  The PIF supports multi-cycle block-read transactions for cache refills and multi-cycle block-write transactions for castouts. Xtensa LX cores can be configured with different cache-line sizes (and block read/write sizes) for instruction or data caches. A cache refill or castout may require 1, 2, 4, 8, or 16 data-bus-width transfers, depending on the PIF data bus width and the cache line size.
- Block-Read Request:
  Block-read requests are only issued for cached line refills when the cache line is larger than a PIF data-bus width. The smallest block size that encompasses an entire cache line is used for each cache-miss request. The instruction and data cache may have different block sizes.
- Block-Write Request:
  Xtensa LX processors only issue block-write requests for dirty-line castouts when they are configured with a writeback cache and their cache line is larger than the PIF data-bus width.
- Write-Buffer Depth:
  A write buffer of 4, 8, 16 or 32 entries is used to mask PIF latency during large bursts of writes, data-cache dirty-line castouts, and register spills.
- Write Responses:
  The Xtensa LX processor can be configured to count write responses for memory-ordering or synchronization purposes. When the write-response configuration option is selected, 16 unique IDs are allocated to write requests, and no more than 16 write requests can be outstanding at any time. In addition, store-release, memory-wait, and exception-wait instructions will wait for all pending write responses to return before those instructions commit. The Xtensa LX processor ignores any write response with a response ID set to a value that does not correspond to an outstanding write request. Consequently, only responses with a matching ID can complete a write transaction waiting on a response.

  When the Xtensa LX processor is not configured to count write responses, all write requests will be issued with the same ID, and store-release, memory-wait, and exception-wait instructions will not wait for write responses before committing. A PIF slave can still send write responses to an Xtensa LX processor, even if write responses were not configured for that processor. An Xtensa LX processor configured without write responses will accept and ignore these responses. Designers may want to design their bridges to always issue write responses for compatibility across multiple processor configurations.

- Write-Bus-Error Interrupt:
  Address and data bus errors reported in write responses do not generate precise exceptions, because the instruction that initiated the store operation would generally have committed in the processor's pipeline by the time a write response could be returned to the processor. Instead, bus errors reported in write responses generate an internal interrupt. This interrupt is maskable by both the interrupt level and interrupt-enable register, and may be configured as a high-priority or level-1 interrupt. The exception virtual address is not available for write-bus errors.
- Read-Conditional-Write Transaction:
  The read-conditional-write transaction implements the optional store-32-bit-compare-conditional instruction (`S32C1I`). The `S32C1I` instruction is a conditional store instruction intended for updating synchronization variables in memory shared among multiple processors or among multiple interrupt handlers. The read-conditional-write request is generated by the Xtensa LX processor for each `S32C1I` instruction. The actual atomic-access mechanism required by the `S32C1I` instruction must be implemented by the PIF slave or the external memory controller.
- Inbound-PIF Request:
  An Xtensa LX core may optionally accept inbound requests from the PIF that accesses the local instruction RAMs, data RAMs, or the XLMI port. The processor must have a PIF for the inbound-PIF Request option to be enabled. The processor must also have at least one instruction RAM, data RAM, or an XLMI port to be compatible with this option. Inbound-PIF requests are configured separately for each configured local memory or XLMI port.
- Inbound-PIF Request Buffer Size:
  When the inbound-PIF request option is enabled, an inbound-PIF request buffer is added to the processor. The depth of the inbound-PIF request buffer limits the maximum block-read or -write request that the Xtensa LX processor can handle. Therefore, the inbound-PIF request buffer size should be configured to be greater than or equal to the largest possible inbound-PIF block-request size. The Xtensa LX processor will not process block requests that are larger than the inbound-PIF request buffer, and will generate an error for inbound-PIF reads or writes that exceed the size of the inbound-PIF request buffer. Note that PIF request and response widths must match for the Xtensa LX PIF implementation.

12.4 PIF Signal Descriptions

All PIF port names are prefixed to indicate the port direction, as well as their role during a request or response transaction. A PIF signal name is constructed by concatenating the port direction, the request or response indicator, and a description of the signal's function. The rules for naming PIF signals are:

- `PO` indicates that a signal is an output of the agent, while `PI` indicates that a signal is an input of an agent.
- `Req` indicates that a signal is involved in the request phase of a transaction, while `Resp` indicates that a signal is involved in the response phase of a transaction.

Thus, all port names follow the scheme: `<PO|PI><Req|Resp><Description>`.

The wire-naming convention used in the Xtensa LX processor test bench is to drop the `<PO|PI>` prefix on the wire names, and prefix the wires with the instance name of the driver of a signal. Request wires are prefixed with the master's instance name, and response wires are prefixed with the slave's instance name. In the following two figures, the <p> and <b> notation represents two different instance names.

Figure 36 shows an example system consisting of a PIF master and a PIF slave. The diagram illustrates the signal and port naming conventions.

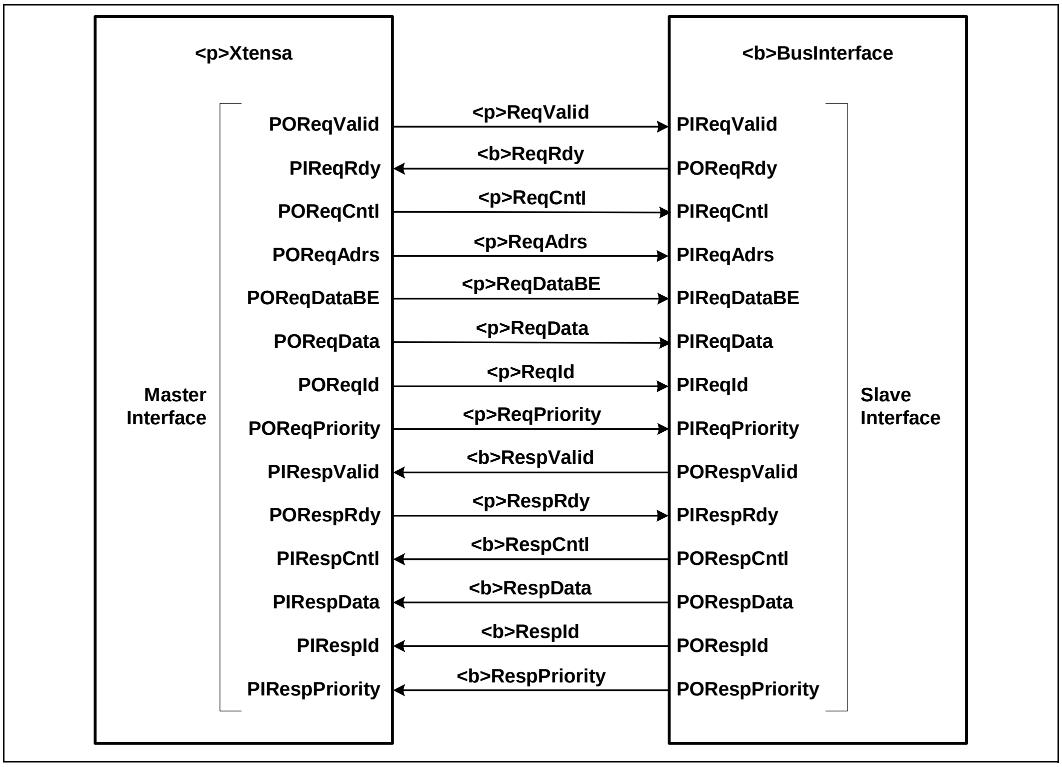


Figure 36. PIF Master and PIF Slave Block Diagram

Figure 37 shows an example system consisting of two PIF master/slaves. A master/slave has a separate master PIF for outbound requests and a slave PIF for inbound requests.

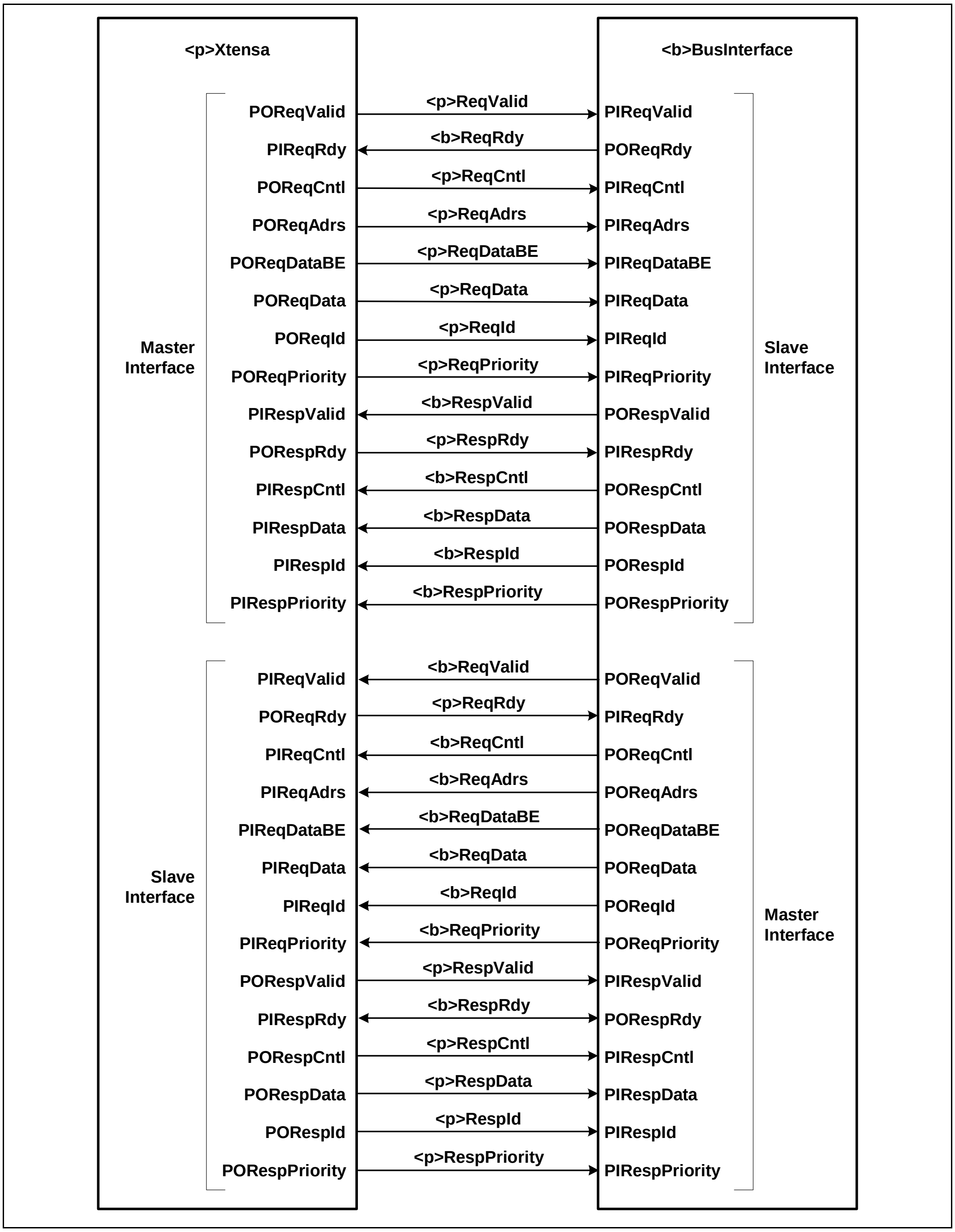


Figure 37. PIF Master/Slave Block Diagram

Table 86 and Table 87 list and describe the PIF interface signals. Components with PIF are categorized as masters, slaves or master/slaves. Table 86 lists the interface signals for a PIF master and Table 87 lists the interface signals for a PIF slave. A PIF master/slave has both sets of interface signals.

Table 86. PIF Master Interface Signals

Name (configured width)	Direction	Description
`POReqValid`	Output	Indicates that there is a valid bus-transaction output request. All other signals prefixed with `POReq` are qualified by `POReqValid`.
`POReqCntl[7:0]`	Output	Encodes the bus-transaction request type and block size for block requests.
`POReqAdrs[31:0]`	Output	Transaction request address. Address is aligned to the transfer size.
`POReqData [pifwidth-1:0]`	Output	Data lines used by requests that require data during the request phase. These requests include single data write, block write, and read-conditional-write requests. The data bus is configurable to 32, 64 or 128-bits, to provide the necessary bandwidth for a given system or application.
`POReqDataBE [pifwidth/8-1:0]`	Output	Byte enables indicating valid lanes during requests that use `POReqData`, or byte lanes expected to be valid during responses that use `PIRespData`.
`POReqId[5:0]`	Output	Request ID. Responses are expected to be tagged with the same ID as the request on `PIRespId`. IDs support multiple outstanding load and store requests. No restrictions or orderings are placed on the usage or uniqueness of IDs.
`POReqPriority[1:0]`	Output	Request priority allows PIF slaves and interconnection networks to make priority judgments when arbitrating bus-transaction requests. 0x0 is low-priority, and 0x3 is high-priority. Xtensa LX processors issue all transactions at priority level 0x2.
`PIReqRdy`	Input	Indicates that the PIF slave is ready to accept requests. A request transfer completes if `PIReqRdy` and `POReqValid` are both asserted during the same cycle.
`PIRespValid`	Input	Indicates that there is a valid response. All other signals prefixed with `PIResp` are qualified by `PIRespValid`.
`PIRespCntl[7:0]`	Input	Encodes the response type and any error status for transaction requests.
`PIRespData [pifwidth-1:0]`	Input	Response data. The data bus is configurable to 32, 64, or 128-bits and is equal in width to the request data bus.
`PIRespId[5:0]`	Input	Response ID. Matches the ID of the corresponding request.
`PIRespPriority[1:0 ]`	Input	Response priority. Matches the priority of the corresponding request. Ignored by Xtensa LX processors.
`POResp Rdy`	Output	Indicates that the PIF master is ready to accept responses. A response transfer completes when `PIRespValid` and `POResp Rdy` are asserted during the same cycle.

Note:

pifwdith = configured PIF width, one of 32, 64, or 128-bits.

Table 87. PIF Slave Interface Signals

Name (configured width)	Direction	Description
`PIReqValid`	Input	Indicates that there is a valid bus-transaction input request. All other signals prefixed with `PIReq` are qualified by `PIReqValid`.
`PIReqCntl[7:0]`	Input	Encodes the bus-transaction request type and block size for block requests.
`PIReqAdrs[31:0]`	Input	Transaction-request address. Address is aligned to the transfer size.
`PIReqData [pifwidth-1:0]`	Input	Data lines used by requests that require data during the request phase. These requests include single data write, block write, and read-conditional-write requests. The data bus is configurable to 32, 64, or 128-bits.
`PIReqDataBE [pifwidth/8-1:0]`	Input	Indicates valid byte lanes during requests that use `PIReqData`, or byte lanes expected to be valid during responses that use `PORespData`.
`PIReqId[5:0]`	Input	Request ID. Responses are expected to be tagged with the same ID as the request on `PORespId`. IDs support multiple outstanding requests. No restrictions or orderings are placed on the usage or uniqueness of IDs.
`PIReqPriority[1:0]`	Input	Request priority allows PIF slaves and interconnection networks to make priority judgments when arbitrating bus-transaction requests. 0x0 is low-priority, and 0x3 is high-priority. Xtensa LX processors issue requests at priority level 0x2.
`POReqRdy`	Output	Indicates that the slave is ready to accept requests. A request transfer completes if `POReqRdy` and `PIReqValid` are both asserted during the same cycle.
`PORespValid`	Output	Indicates that there is a valid response. All other signals prefixed with `POResp` are qualified by `PORespValid`.
`PORespCntl[7:0]`	Output	Encodes the response type and any error status on requests.
`PORespData [pifwidth-1:0]`	Output	Response data. The data bus is configurable to 32, 64, or 128-bits and is equal in width to the request data bus.
`PORespId[5:0]`	Output	Response ID. Matches the ID of the corresponding request.
`PORespPriority[1:0 ]`	Output	Response priority. Matches the priority of the corresponding request. Ignored by Xtensa LX processors.
`PIRespRdy`	Input	Indicates that the master is ready to accept responses. A response transfer completes when `PORespValid` and `PIRespRdy` are asserted during the same cycle.

Note:

pifwdith = configured PIF width, one of 32, 64, or 128-bits.

13. Xtensa LX System Signals and Interfaces

This section describes additional Xtensa LX signals and interfaces:

- System-level signals (see Section 13.2 on page 166)
  - clock
  - reset
  - wait mode
  - run/stall
  - processor ID signals
- On-chip debug (see Section 13.3 on page 169)
  - interface to JTAG TAP controller
- Test access and trace port (see Section 13.4 on page 170)
  - status and instruction data output
- Interrupt signals (see Section 13.5 on page 171)
  - interrupt pins
  - interrupt options

13.1 Xtensa LX Configuration Options

The following configuration options are covered by this section:

- Processor ID:
  Optional 16-bit input to give the processor a unique ID for multi-processor systems. The Processor ID option is available to Xtensa LX licensees who have purchased the System Modeling and Multiprocessor Support package.
- Interrupts and High-Priority Interrupts:
  The Xtensa LX processor can have as many as 32 interrupt signals from external sources. Each interrupt line can be configured as level- or edge-triggered.
- Non-maskable Interrupt:
  The Xtensa LX processor can have one Non-Maskable Interrupt (NMI) signal.
- On-Chip Debug:
  An Xtensa LX core may optionally include an On-Chip Debug (OCD) module to interface with a JTAG TAP controller.
- Trace Port:
  The Xtensa LX processor can output status and instruction data on a separate, optional trace bus.

13.2 Xtensa LX System Signals

Table 88 provides a description of the Xtensa LX system signals.

Table 88. System Signals

Name	Direction	Config Option	Description
`CLK`	Input	Any	Clock input to the processor (duty cycle must fall between the ratios of 60/40 and 40/60). Note: `CLK` is used to generate the internal processor clock signal `PCLK`.
`BReset`	Input	Any	Reset input to the processor. This signal may be asserted asynchronously, but must be negated synchronously with `PCLK`. The reset signal is registered for two cycles before it is used internally. `BReset` is active high. A minimum of 50 `PCLKs` are required to reset the Xtensa LX processor.
`PWaitMode`	Output	Interrupt	Indicates that the processor is in sleep mode. The processor asserts this signal when it has executed a `WAITI` instruction and is waiting for an interrupt. Any asserted interrupt that is not disabled will wake the processor, which will then jump to the appropriate designer-configured interrupt vector.
`RunStall`	Input	Any	This external signal stalls the processor. This signal passes through a flip-flop within the Xtensa LX processor before it is used.
`PRID[15:0]`	Input	PRID	Input to the processor, latched at reset into the low-order bits of the PRID (processor ID) special register. The PRID register option is available to Xtensa LX licensees who have purchased the System Modeling and Multiprocessor Support package.

13.2.1 CLK

The `CLK` signal is the input clock to the Xtensa LX processor. This clock is gated and distributed throughout the Xtensa LX core. The Xtensa LX clock tree generates `PCLK`, which represents a leaf node of the non-gated clock-distribution tree.

13.2.2 Reset

The Xtensa LX processor registers the `BReset` signal for two clock cycles, which incurs a reset latency. The system design must take this 2-cycle reset delay into account to synchronize processor resets with the rest of the system logic. The reset signal must be asserted continuously for at least 50 clock cycles.

Note that all busy signals from the instruction-RAM, data-RAM, and XLMI ports as well as the run/stall signal (see Section 13.2.4 on page 167) can be active during and immediately following the assertion of reset. This feature can be useful for initializing RAMs at reset time.

The reset input to the processor can be asserted asynchronously, but must be negated synchronously with `PCLK`. The reset signal is registered for two cycles before it is used internally. `BReset` is active high. A minimum of 50 `PCLK`s are required to reset the Xtensa LX processor.

13.2.3 Wait Mode

The wait mode option adds the `PWaitMode` port and `WAITI` instruction to the processor core. Wait mode is always enabled as part of the interrupt option. When the `WAITI` instruction completes, it sets the interrupt level in the processor's `PS.INTLEVEL` register to the value encoded in the instruction, waits for all of the processor memory interfaces to become idle, and causes the processor to assert the `PWaitMode` signal. Processor operations suspend until a non-masked interrupt occurs.

If the global clock-gating option is enabled, then the processor generator will add gating to the majority of the clocks in the processor. Clock gating saves power while `PWaitMode` is asserted. If global clock gating is not enabled, then processor operations will be suspended while `PWaitMode` is asserted, but no power-saving mechanism will be active inside the Xtensa LX processor.

Inbound-PIF requests are not considered processor operations and continue while the processor is in wait mode. The inbound-PIF-request logic's clock network is gated by a separate set of conditions from most of the other Xtensa LX processor logic, to allow inbound-PIF requests to continue functioning while the processor is in wait mode.

13.2.4 Run/Stall

The `RunStall` signal allows an external agent to stall the processor and shut off much of the clock tree to save operating power.

Global Run/Stall Behavior

The `RunStall` input allows external logic to stall the Xtensa LX processor's internal pipeline. Any in-progress PIF activity such as instruction- or data-cache line fills will complete after the assertion of the `RunStall` input line and inbound-PIF requests to the local instruction and data RAMs, and the XLMI port will still function even though the pipeline has stalled.

The `RunStall` input has two intended uses:

- It can be used as a mechanism to initialize local instruction and data RAMs using inbound-PIF requests after a reset. If the `RunStall` input is asserted during reset and remains asserted after the reset is removed, the processor will be in a stalled state. However inbound-PIF requests to local instruction and data RAMs and the XLMI port can still occur. The Instruction and data RAMs can be initialized using inbound-PIF requests and after these RAMs have been initialized, the `RunStall` can be released, allowing the processor to start code execution. This mechanism allows the processor's reset vector to point to an address in local instruction RAM, and the reset code can be written to the instruction RAM by an external agent using inbound-PIF write requests before the processor starts executing code.
- The `RunStall` input can be asserted at any time during processor operation to freeze the internal pipeline. Assertion of `RunStall` reduces the processor's active power dissipation without using or requiring interrupts and the `WAITI` option. However, the power savings resulting from use of the `RunStall` signal will not be as great as for those using `WAITI`, because the `RunStall` logic allows some portions of the processor to be active even though the pipeline is stalled. (See Chapter 22, "Low-Power SOC Design Using Xtensa LX" on page 221 for more information on low-power design using the Xtensa LX processor.)

`RunStall` signal timing appears in Figure 38. The processor synchronizes the `RunStall` input internally so that the pipeline stalls one cycle after `RunStall` is asserted. The `GlobalStall` signal shown in the diagram is an internal processor signal that gates much of the processor's clock tree. The resulting gated clock appears as `G1SCLK` in the figure. The Xtensa LX pipeline restarts two clock cycles after `RunStall` is negated.

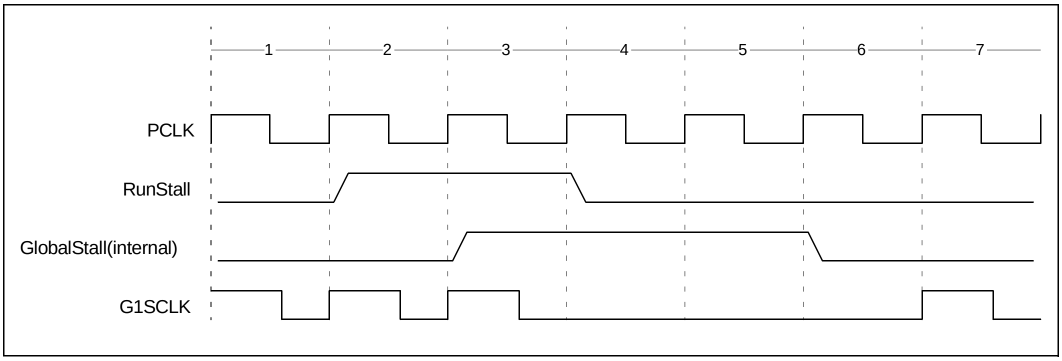


Figure 38. RunStall Behavior and Timing

The extra stall cycle following the negation of the `RunStall` input allows reactivation of the local-memory enables. To save power, memory enables are deactivated during `RunStall`-initiated pipeline stalls unless one of these two conditions exists:

- A cache-line fill was in progress when `RunStall` is asserted.
- Inbound-PIF requests are being made to local instruction or data RAMs.

13.2.5 Processor ID Option

Some applications employ multiple Xtensa LX processors that execute from the same instruction memory so there is often a need to distinguish one processor from another in these situations. The processor ID (PRID) option allows the system logic to provide each Xtensa LX processor with a unique identity by adding a 16-bit input bus, `PRID`, as an input to the Xtensa LX core. When the processor comes out of reset, this input is latched into the low 16-bits of special register 235, which can be read by an `RSR.PRID` instruction.

The `PRID` port is not a dynamic input. The Xtensa LX processor only latches the value of the `PRID` port when `BReset` is negated and the processor exits the reset state. The `PRID` inputs must be stable for 10 clock cycles before and 10 clock cycles following the negation of `BReset`. Changing the `PRID` inputs within that 20-clock period produces unpredictable results. Changing the `PRID` input more than 10 cycles after the negation of reset has no effect on the PRID register.

Notes:

- The PRID register option is available to Xtensa LX licensees who have purchased the System Modeling and Multiprocessor Support package.
- Future Xtensa LX implementations may have wider or narrower `PRID` input ports.

13.3 Debug and Status Signals

The signals in this section comprise the debug interface of the Xtensa LX processor. Section 13.4 “Test Access Port Interface” on page 170 describes the debug functionality. Table 89 lists and describes the debug and status signals.

Table 89. Debug Status Signals

Name (configured width)	Direction	Config Option	Description
JTCK	Input	OCD	Standard 1149.1 JTAG TAP clock.
JTDI	Input	OCD	Standard 1149.1 JTAG TAP port serial data in.
TResetB	Input	OCD	Reset signal from TAP Controller. (Asserted LOW).
TShiftDR	Input	OCD	TAP state machine in Shift-DR state.
TClockDR	Input	OCD	Standard 1149.1 JTAG TCK when the TAP state machine is in Capture-DR or Shift-DR state.
TUpdateDR	Input	OCD	TAP state machine in Update-DR state.
TUpdateIR	Input	OCD	TAP state machine in Update-IR state.
TInst [4:0]	Input	OCD	Instruction inputs into the TAP Instruction Register.
XTDO	Output	OCD	Test Data Output from the processor's JTAG Data Registers.
XTDV	Output	OCD	Indicates that the Test Data Output (XTDO) from the processor is valid.
XOCDMode	Output	OCD	Indicates that the processor is in OCD halt mode.
XOCDModePulse	Output	OCD	A 1-cycle pulse from the processor indicating that it has entered OCD halt mode.
TDebugInterrupt	Input	OCD	External debug interrupt.
PStatus [pstatuswidth-1:0] (See notes)	Output	Traceport	Processor debug status during W stage.
PDebugData [pdebugwidth-1:0] (See notes)	Output	Traceport	Processor debug data during W stage.
TMode	Input	Scan	Special test mode signal for scan testing.

Notes:
pstatuswidth =8 if only address trace is configured and 16 if data trace is configured
pdebugwidth= 32 if only address trace is configured and 64 if data trace is configured

13.4 Test Access Port Interface

If the OCD configuration option is selected, the Xtensa LX processor incorporates an on-chip debug module that interfaces directly to an IEEE 1149.1-compliant, TAP-controller interface, which was designed and is supplied by Tensilica as part of the Xtensa LX support package.

13.4.1 Trace Port

The Xtensa LX processor optionally provides status outputs that reflect the processor's internal state during normal operation. This trace port’s main function is to enable program tracing. Optionally, it is possible to trace the flow of data for all Xtensa LX core instructions using the trace port.

Note: The trace port provides only minimal visibility for designer-defined TIE instructions.

The PDebugData and PStatus signals comprise the trace port.

PDebugData:

This port provides information on instructions, data and loop-begin virtual addresses, Xtensa LX AR register file write data, and additional operation details for certain pipeline bubbles.

Notes:

- Writes to TIE register files are not traced.
- Writes made by long-latency TIE instructions to the processor’s AR register file that occur after the W stage are also not traced.
- If the AR register file is configured with multiple write ports, only the port that performs writes for core processor instructions is traced.
- For configurations with multiple load-store units, only the unit that performs loads and stores for core instructions is traced.

PStatus:

`PStatus` indicates the status of the instruction currently in the processor’s W stage (or it indicates that there is no instruction in the W stage) and the cause of a pipeline bubble.

13.5 Interrupt Features

13.5.1 Level-Triggered Interrupt Support

Level-triggered interrupts are set and cleared by the external source of the interrupt. The time required for the processor to take an interrupt varies widely and depends on many factors. Therefore, external logic that generates level-triggered interrupts should assert and hold the interrupt line active until the interrupt is serviced by the software.

Note: The Xtensa LX interrupt-vector model is such that many interrupts can share the same interrupt vector and service handler.

13.5.2 Edge-Triggered Interrupt Support

A single-cycle pulse synchronous to `PCLK` on an interrupt line from an external source signals an edge-triggered interrupt. The processor latches the interrupt on the rising edge of `PCLK`. The edge-triggered interrupt condition is cleared by writing to a special register.

13.5.3 NMI Interrupt Support

One interrupt input can be designated as the non-maskable interrupt. This interrupt is sampled like an edge-triggered interrupt as described above but it cannot be masked and has a separate interrupt vector (as do high-priority interrupts).

Because the Xtensa LX processor does not mask the NMI, the time period between successive assertions of NMI should not be so short as to interrupt the processor while it is saving state from a previous NMI. Such an occurrence will prevent the processor from being able to return to the user code that was executing when the first NMI was triggered. When calculating the amount of time the processor needs to save its state prior to starting an NMI service routine, the designer should include memory-system latencies, which will affect the time required to store processor-state values. For processes or events that will generate frequent interrupts, the designer should use a high-level interrupt instead of the NMI and should assure that the interrupt is properly unmasked as soon as the absolute required state has been saved.

13.5.4 Example Core to External Interrupt Mapping

An example of Xtensa LX core-to-external interrupt signal mapping is illustrated in Table 90. `BInterrupt[0]` is associated with the lowest order bit of the interrupt register that is configured as an external interrupt (an edge-triggered interrupt, level-triggered interrupt, or a non-maskable edge-triggered interrupt). For the example shown below, bit 1 is the lowest-order bit in the interrupt register to be associated with an external interrupt because bit 0 has been configured as a software interrupt. `BInterrupt` input lines with increasingly higher numbers are associated with increasingly significant bits in the interrupt register, if those bits are configured for external interrupts.

In this example, bit 2 of the interrupt register is assigned to one of the Xtensa LX processor's internal timers, so the `BInterrupt[1]` interrupt input is associated with the next-most significant bit assigned to an external interrupt, which in this example is bit 3 of the interrupt register. Similarly, the `BInterrupt[2]` interrupt input is associated with bit 4 of the interrupt register.

Note: Numerous other interrupt mappings are possible. Table 90 merely shows one possible configuration as an illustration.

Table 90. Example Core to PIF External Interrupt Mapping

Interrupt/Enable Bit	Interrupt Type	BInterrupt Bit
0	Software	
1	External Edge	`BInterrupt[0]`
2	Timer	
3	External Level	`BInterrupt[1]`
4	External Edge	`BInterrupt[2]`
5	Software	

13.6 Interrupt Handling

The Xtensa LX processor can have as many as 32 level- or edge-triggered interrupt sources. The states of these interrupt sources are latched in a special 32-bit register, named the `INTERRUPT` register. Each `INTERRUPT` register bit can be independently configured to indicate external, software, or Internal interrupt classes.

External interrupts are connected to the `BInterrupt` input pins and can be configured to be of three types:

- Level-sensitive
- Edge-triggered
- NMI (edge-triggered non-maskable interrupt)

Note: Only one NMI interrupt signal is allowed.

Software interrupt bits are readable and writable by software, which uses the `INTERRUPT` register address to read, set, and clear a software interrupt.

Internal interrupt bits are set by processor hardware (that is, timer logic) and can only be read by software, which can clear the interrupt indirectly by writing to the associated source of the interrupt (that is, a timer control register).

Note: See Section 15.10 for information about Xtensa LX bus errors and associated exception behavior.

External logic that generates interrupts should be synchronized with `PCLK` (the internal processor clock) so that they meet the setup-time requirements.

Level-sensitive interrupt inputs are latched directly in the `INTERRUPT` register. The processor will continue taking the associated interrupt exception as long as a `BInterrupt` level-sensitive input is asserted. Like the internal interrupts, external level-sensitive interrupts must be cleared at the source.

Edge-triggered and NMI interrupt inputs are synchronized by an internal processor register, clocked by PCLK, and processed by edge-detector logic that detects low-to-high signal level changes. Edge-triggered interrupt inputs must be asserted for at least one one processor clock cycle to ensure detection. After the corresponding interrupt bit is set, it will remain set even when the external signal is negated. Software can read these edge-triggered interrupt indicator bits by reading the INTERRUPT register and can clear these bits by writing to the INTCLEAR register (special register 227).

Note: Software cannot set external interrupt bits in the INTERRUPT register.

Interrupt-enable bits are defined by another special register, named the INTENABLE Register, which is readable and writable. The INTERRUPT register bits are bitwise AND'ed with the corresponding bits of the INTENABLE register. The resulting bits are then all OR'ed together to form an internal interrupt signal that can cause an Xtensa LX interrupt exception.

To reiterate, edge-triggered, internal, and software interrupt sources can set the associated interrupt bits in the INTERRUPT register, but only software can clear these bits using the INTCLEAR register. Level-sensitive interrupt sources directly set and clear the associated interrupt bits in the INTERRUPT register.

Designers can also configure any of the interrupt signals to be a NMI (non-maskable interrupt). A bit in the INTERRUPT register designated as the NMI bit will not be masked by the corresponding INTENABLE register bit.

14. Local-Memory Usage and Options

Local memory for the Xtensa LX processor consists of caches, local data memory, and local instruction memory. The processor also has a fast local memory port (XLMI) that can be used to connect the Xtensa LX processor to additional local memories or fast, memory-mapped peripheral blocks.

14.1 Cache Organization

Caches are fast and small memories that are used to buffer the processor from slower and larger memories located external to the SOC. The Xtensa LX core's caches store the data and instructions that a program is currently using, while the rest of the data resides in slower main memory external to the Xtensa LX core. In general, the Xtensa LX processor can access the separate instruction and data caches simultaneously to maximize cache-memory bandwidth.

The Xtensa LX processor accesses cache information when there is a *hit* in the cache look-up sequence. Xtensa LX caches are physically tagged. A hit occurs when there is a match between one of the cache's physical-address tags and the physical target address. A *miss* occurs when the physical address tags of the cache do not match the physical target address. A miss means that the target instruction or data is not present in the cache and that the cache line must be retrieved from the slower external memory.

A cache line is one entry in the cache (per way for set-associative caches). For example, each way of a 4-way set associative cache has its own cache line, which is indexed and used if hit and refilled if a load or store causes a cache miss.

When there is a cache hit, the Xtensa LX processor fetches the target instruction or data from the appropriate cache and the operation continues. A read cache miss initiates a cache-line refill sequence involving a pipeline replay and the generation of a request for new data from external memory before the operation can continue.

14.1.1 Organization of Instruction Cache

The instruction cache has the following configurable characteristics:

- cache size
- cache-line size
- set-associativity
- virtually indexed
- checked with a physical tag

- lockable on a per-line basis

The instruction-cache tag-array width depends on the configured data-array size, the choice of set associativity (line replacement algorithm bit appears depending on configuration), and the presence or absence of line locking.

Note: Although the instruction-cache data array actually holds instructions, the array is referred to as a data array to distinguish it from the instruction cache's tag-field array.

14.1.2 Organization of Data Cache

The data cache has the following configurable characteristics:

- cache size
- cache-line size
- set associativity
- write-through or write-back cache write policy
- virtual address indexing
- checked with a physical tag
- lockable on a per-line basis

The data-cache tag array width depends on the configured data array size, the choice of set associativity (the line-replacement-algorithm bit appears or disappears depending on configuration), the cache write-policy choice (a dirty bit is present when write-back is configured), and the presence or absence of line locking.

14.2 Region Protection Unit (RPU)

The Xtensa LX processor implements a simple region-protection scheme that provides access protection for eight evenly sized 512-Mbyte regions in the processor's address space. The region-protection field in the Region Protection Unit (RPU) can allow or deny access to a region, turns caching for the region on or off, and sets the region's caching type (write-through or write-back). The processor accesses the RPU in parallel with its local memories and the RPU generates any protection exceptions that may occur.

14.4 RAM, ROM, and XLMI organization

If configured, each instruction RAM or ROM, each data RAM or ROM, and the XLMI port are assigned a physical base address at configuration time. This physical base address must be aligned to a multiple of the size of the RAM, ROM, or XLMI port. The processor uses these base addresses to detect when a specific access falls within a memory space of one of the local memories, or the XLMI port.

There can be no overlap among the local memories or the XLMI port.

As described earlier, the Xtensa LX processor can have one or two load/store units. Each load/store unit can independently perform loads or stores. Each data RAM, data ROM, and XLMI port must be connected to all configured load/store units. The base addresses of the local memories and the XLMI port are the same for all configured load/store units.

Data memories connected to two load/store units must be able to deal with access requests from both load/store units by being multi-ported, for example. The same is true for the XLMI port.

14.5 Memory Combinations, Initialization, and Sharing

14.5.1 Combinations of Cache, RAM, ROM, and the XLMI port

The Xtensa LX processor can be configured with a maximum of four local data-memory and four local instruction-memory ports. Local data-memory ports include the data-RAM ports, the data-ROM port, the XLMI port, and ports for data-cache ways. Because each way of the data cache requires a separate interface port (each of the cache's ways is a separate data array), a three-way data cache, for example, consumes three of the four available local data ports and therefore allows configuration of only one additional local data memory. Xtensa LX configurations with two load/store units cannot have a data cache, but can have a data RAM, a data ROM, and an XLMI port.

Local instruction-memory interfaces include the instruction-RAM ports, the instruction-ROM port, and one way of an instruction cache. There can be at most two instruction-RAM ports and two data-RAM ports. There can be at most one instruction-ROM port, one data-ROM port, and one XLMI port.

For each cache, the number of interfaces—and the number of data/tag memory array pairs—depends on the configured associativity. The maximum allowed cache associativity is four, therefore, the number of cache interfaces (for the data/tag array pairs) can be any integer from zero to four.

Note: The Xtensa LX processor must be configured with at least one local data-memory interface and one local instruction-memory interface.

14.6 Memory-Access Restrictions and Exceptions

The local memory interfaces are optimized for either data memories or instruction memories. Instruction memory interfaces include the instruction RAM and ROM ports. The instruction memory interface is designed for fast instruction fetches, but is not optimized for load or store operations. If a 32-bit load operation accesses an instruction memory, the Xtensa LX processor will flush its pipeline and will replay the load. If a 32-bit store operation accesses an instruction memory, the Xtensa LX processor will flush its pipeline and replay the instruction following the store. Consequently, firmware literals should be placed in data memory—not instruction memory—to avoid significant performance reduction.

15. TIE for Hardware Developers

15.1 Programmable Processors as RTL Replacement

Most System-On-a-Chip (SOC) designs consist of one or more programmable processors, multiple memory cores, and numerous RTL modules. In a typical design, the performance-critical functions are implemented in RTL, whereas control-oriented functions that need more flexibility are coded in software running on a programmable processor. In many situations, designers need both the performance of custom RTL design and the flexibility of a programmable solution. Creating application-specific extensions to the Xtensa LX microprocessor core using the TIE language may provide a very attractive solution for such designs.

The TIE language allows you to create new register files, computational units, and custom data paths that significantly enhance the Xtensa LX processor's computational bandwidth. The Xtensa LX processor offers the traditional memory and bus interfaces offered by most processors, but the TIE language allows you to create new, custom interfaces between the processor and other pieces of SOC hardware. The TIE design methodology allows you to develop a design solution using correct-by-construction design techniques that has both the flexibility of a programmable processor and the performance of a custom RTL implementation.

This methodology thus allows you to use processors in many design situations where custom RTL design was previously the only practical approach. The advantage of this approach is that, by using a configurable processor and the TIE language, you can develop these solutions using a pre-verified IP core as a foundation and add custom extensions through correct-by-construction design techniques. This design approach significantly reduces the need for the long verification times required when designing custom RTL.

The following sections provide an overview of some of the major capabilities of the TIE language.

15.2 A Simple Hardware Extension

Consider a simple computation that calculates the average of two 16-bit numbers. On most general-purpose programmable processors, this computation can be performed by executing two instructions: an “add” and a “shift right” of the sum by 1. For a hardware designer implementing such an operation in RTL, it is easy to see that the “shift right by 1” is essentially a free operation (it requires no gates, only wires), and thus the entire computation can be performed in about the same amount of time as the addition, and with the same number of gates. Using TIE, you can create a new instruction (a hardware execution unit) that performs this computation as follows:

```
operation AVERAGE {out AR res, in AR input0, in AR input1} {} {
     wire [16:0] tmp = input0[15:0] + input1[15:0];
     assign res = tmp[16:1];
}
```

The above instruction takes two input values (`input0` and `input1`) from the Xtensa LX processor's AR register file to compute its output, `res`. This result is then put back into the AR register file. The semantics of the instruction, an add followed by a right shift, are described using standard Verilog-like syntax. This 3-line description is all you need to create a new processor instruction that computes the average of two 16-bit numbers. From this description, the TIE compiler automatically generates the data path, pipeline and control hardware necessary for the processor to execute this instruction.

15.3 Creating new Registers

The above example illustrates an instruction that operates on values stored in the Xtensa LX processor's AR register file, which is a 32-bit-wide register file. Now consider the problem of implementing a multiply/accumulate operation that multiplies two 16-bit numbers to generate a 32-bit product, which must be accumulated in a 40-bit accumulator. You need a 40-bit register for the accumulator, but most general-purpose 32-bit processors lack such wide registers. TIE allows you to create new registers (or "state" as they are referred to in the TIE language) that can then be used to store input or output operands of designer-defined instructions. The following piece of code implements the multiply-accumulate instruction as described above.

```
state ACCUM 40 add_read_write

operation MAC {in AR m0, in AR m1} {inout ACCUM} {
     assign ACCUM = TIEmac(m0[15:0], m1[15:0], ACCUM, 1'b1, 1'b0);
}
```

The first line of code declares a new 40-bit state (or register) named `ACCUM`. The `add_read_write` flag is optional and, if present, generates instructions that move data between the AR register file and the new state register. These instructions can be used to initialize the new accumulator, or to copy the accumulator contents into an AR register.

The `MAC` instruction takes two inputs (`m0` and `m1`) from the AR register file and generates an output that is stored in the `ACCUM` state register. TIEmac is a built-in TIE module that takes five arguments. The first two are the multiplier and multiplicand, respectively. The third argument is the accumulator. The fourth argument is a 1-bit flag that indicates whether the multiplication is signed or unsigned, and the fifth argument is a flag that indicates whether the product is to be added to the accumulator, or subtracted from it. Thus, TIEmac provides a convenient, short-hand notation for specifying a multiply/accumulate operation with different options.

15.4 Multi-cycle Instructions

The `MAC` example above implements data path hardware that performs 16-bit multiplication with 40-bit accumulation. For designs that have an aggressive clock-frequency goal, it is possible that this computation may not complete in one processor clock cycle because multipliers are generally slower than other sorts of logic constructs used in processor pipeline stages. To prevent slower operations from degrading the processor's maximum clock speed, the TIE language allows you to spread computations over multiple clock cycles using the *schedule* construct.

The schedule construct specifies the pipeline stage in which the input operands of the computation are used (*use*) and output operands defined (*def*). Instructions with all operands having the same use/def stage are single-cycle instructions. Their input operands are available to the processor's data path logic at the beginning of the specified cycle, and their result is expected by the end of the same cycle. Instructions with one or more output operands that have a def stage later than one or more of the input operands are multi-cycle instructions. For example, the following schedule for the MAC instruction makes it a 2-cycle instruction:

```
schedule sch_mac {MAC} {
     use m0 1;
     use m1 1;
     use ACCUM 2;
     def ACCUM 2;
}
```

The above schedule specifies that the multiplier (`m0`) and multiplicand (`m1`) are used in stage 1, whereas the accumulator (`ACCUM`) is used and defined in stage 2. The hardware thus has two clock cycles to perform the computation, which should be an adequate amount of time for most SOC designs.

From the operation and schedule description of the MAC instruction, the TIE compiler implements all the hardware necessary for the multiply/accumulate data path, as well as control logic to interface this database to the processor's pipeline. The data path is implemented in a fully pipelined manner, which means that it's possible to issue a MAC instruction every cycle even though it requires two cycles for it to compute the result.

Previous releases of the TIE compiler required use of the behavioral retiming feature of synthesis tools to synthesize multi-cycle TIE instructions. The Xtensa LX version of the TIE compiler allows the designer more control over multi-cycle TIE instruction implementations. This enhancement to the TIE compiler may alleviate the need for behavioral retiming by allowing the designer to schedule wire usage.

15.5 Wide Data Paths and Execution Units

The base Xtensa processor contains a 32-bit register file and a 32-bit memory interface. Many computational tasks require processing data whose native size is not 32-bits. Some tasks require a wider memory interface because of the sheer bandwidth requirement of the application. Processing such data on generic 32-bit processors is cumbersome and time consuming, because each native data transfer and computation must be decomposed into 32-bit chunks.

The TIE language allows you to create wide register files and a wide memory interface. It also allows you to create execution units that directly operate on the values stored in these wide register files.

Consider the problem of processing 8-bit video or pixel data. Assume that you need to process eight pixels in one cycle to meet your overall system-throughput goals. An example piece of TIE code for designing such hardware is as follows:

```
regfile VR 64 8 v

operation VADD {out VR sum, in VR in0, in VR in1} {} {
     wire [7:0] sum0 = in0[ 7: 0] + in1[ 7: 0];
     wire [7:0] sum1 = in0[15: 8] + in1[15: 8];
     wire [7:0] sum2 = in0[23:16] + in1[23:16];
     wire [7:0] sum3 = in0[31:24] + in1[31:24];
     wire [7:0] sum4 = in0[39:32] + in1[39:32];
     wire [7:0] sum5 = in0[47:40] + in1[47:40];
     wire [7:0] sum6 = in0[55:48] + in1[55:48];
     wire [7:0] sum7 = in0[63:56] + in1[63:56];

     assign sum = {sum7, sum6, sum5, sum4, sum3, sum2, sum1, sum0};
}
```

The regfile construct defines a new register file that is 64 bits wide and has eight entries. Each of these entries can be viewed as a vector consisting of eight 8-bit pixels. The VADD operation creates an execution unit that performs eight 8-bit additions in parallel. Additional vector instructions can be defined in a similar manner to create a vector-processing engine that is capable of performing a variety of operations on eight pixels in one instruction. Such machines are commonly referred to as Single Instruction Multiple Data (SIMD) machines.

Because the VR register file is 64 bits wide, it is also appropriate to configure the Xtensa processor to have a 64-bit-wide memory interface. This wider memory interface lets the processor load and store entire 64-bit vectors in one memory access, thus doubling the memory bandwidth compared to a standard 32-bit memory interface. Note that the Xtensa LX processor's memory interface can be configured to be 32, 64, or 128 bits wide.

15.6 TIE Functions

A TIE function is similar to a function in Verilog. It encapsulates some computation, which can then be used in different places in a TIE description. TIE functions can be used to make the code modular, by providing one description of a computation that gets instantiated in multiple places. It can also be used to share hardware between multiple semantic blocks by declaring the function to be shared. The following piece of TIE code provides an alternative implementation of the ADD16/SUB16 example of Section 25.6 of the *Xtensa LX Microprocessor Data Book*:

```
function [15:0] as16 ([15:0] a, [15:0] b, add) {
     wire [15:0] tmp = add ? b : ~b;
     wire carry_in = add ? 1'b0 : 1'b1;

     assign as16 = TIEadd(a, tmp, carry_in);
}
function [15:0] addsub0 ([15:0] a, [15:0] b, add) shared {
     assign addsub0 = as16(a, b, add);
}
function [15:0] addsub1 ([15:0] a, [15:0] b, add) shared {
     assign addsub1 = as16(a, b, add);
}
semantic add16 {ADD16} {
     wire [15:0] sum0 = addsub0(in0[15: 0], in1[15: 0], 1'b1);
     wire [15:0] sum1 = addsub1(in0[31:16], in1[31:16], 1'b1);

     assign res = {sum1, sum0};
}
semantic sub16 {SUB16} {
     wire [15:0] diff0 = addsub0(in0[15: 0], in1[15: 0], 1'b0);
     wire [15:0] diff1 = addsub1(in0[31:16], in1[31:16], 1'b0);

     assign res = {diff1, diff0};
}
```

In this example, `as16` is a function that performs a 16-bit addition or subtraction based on the 1-bit function argument `add`. The primary purpose of this function is to encapsulate the hardware that performs the addition and subtraction. Every instantiation of this function will result in a new and unique copy of the hardware that implements the function.

Functions `addsub0` and `addsub1` are defined as shared functions. There is only one instance of the hardware corresponding to this function and it is shared by all instantiations of the function. These functions are used in the semantic description for both the `ADD16` and `SUB16` instructions. The TIE compiler automatically generates the data path and control logic to share the `addsub0` and `addsub1` hardware between the `ADD16` and `SUB16` instructions.

The semantic descriptions of this section look very similar to the operation descriptions in Section 25.6 of the *Xtensa LX Microprocessor Data Book*. Hardware sharing is expressed through the use of shared functions instead of writing a common semantic for the two instructions and explicitly specifying the operand multiplexing inside this semantic. Thus, shared functions simplify the specification of instructions that share hardware.

15.7 FLIX - Flexible-Length Instruction Xtensions

FLIX (Flexible-Length Instruction Xtensions) is a powerful TIE feature that allows you to create multiple execution units that operate in parallel and instruction formats with multiple opcode slots. Each opcode slot can hold a combination of existing Xtensa LX instructions and new, designer-defined instructions. FLIX instruction formats can be either 32- or 64-bits wide.

Long instructions allow more encoding freedom, where a large number of sub-instruction or operation slots can be defined (although three to six independent slots are typical) depending on the operational richness required in each slot. The operation slots need not be equally sized. Big slots (20-30 bits) accommodate a wide variety of opcodes, relatively deep register files (16-32 entries), and three or four register-operand specifiers. Developers should consider creating processors with big operation slots for applications with modest degrees of parallelism but a strong need for flexibility and generality within the application domain.

Small slots (8-16 bits) lend themselves to direct specification of movement among small register sets and allow a large number of independent slots to be packed into a long instruction word. Each of the larger number of slots offers a more limited range of operations, fewer specifiers (or more implied operands), and shallower register files. Developers should consider creating processors with many small slots for applications with a high degree parallelism among many specialized function units.

The following TIE code is a short but complete example of a very simple long-instruction word processor described in TIE with FLIX technology. It relies entirely on built-in definitions of 32-bit integer operations, and defines no new operations. It creates a processor with a high degree of potential parallelism even for applications written purely in terms of standard C integer operations and data-types. The first of three slots supports all the commonly used integer operations, including ALU operations, loads, stores, jumps and branches. The second slot offers loads and stores, plus the most common ALU operations. The third slot offers a full complement of ALU operations, but no loads and stores.

```
format format1 64 {base_slot, ldst_slot, alu_slot}

slot_opcodes base_slot {ADD.N, ADDX2, ADDX4, SUB, SUBX2, SUBX4,
ADDI.N, AND, OR, XOR, BEQZ.N, BNEZ.N, BGEZ, BEQI, BNEI, BGEI, BNEI,
BLTI, BEQ, BNE, BGE, BLT, BGEU, BLTU, L32I.N, L32R, L16UI, L16SI, L8UI,
S32I.N, S16I, S8I, SLLI, SRLI, SRAI, J, JX, MOVI.N }

slot_opcodes ldst_slot { ADD.N, SUB, ADDI.N, L32I.N, L32R, L16UI,
L16SI, L8UI, S32I.N, S16I, S8I, MOVI.N }

slot_opcodes alu_slot {ADD.N, ADDX2, ADDX4, SUB, SUBX2, SUBX4, ADDI.N,
AND, OR, XOR, SLLI, SRLI, SRAI, MOVI.N }
```

The first line of the example declares a new 64-bit-wide FLIX instruction format containing three operation slots: the base_slot, the ldst_slot, and the alu_slot. The second line lists all the instructions that can be packed into the first of those slots: base_slot. In this example, all the instructions happen to be pre-defined Xtensa LX instructions. However, new TIE instructions can also be included in this slot.

The processor generator automatically creates a NOP (no operation) for each FLIX operation slot, so the software tools can always create complete FLIX instructions, even when no other operations for an operation slot are available for packing into a long instruction. The last two lines of the example designate the subset of instructions that can go into the other two operation slots.

Thus FLIX allows the generation of multiple execution units that can operate in parallel, resulting in higher computational throughput. FLIX technology may be especially useful when using the Xtensa LX processor to implement algorithms and tasks that previously could only be performed by hard-wired RTL blocks.

15.8 TIE Ports and Queues

General-purpose microprocessors communicate with external devices using a set of interface signals collectively referred to as a bus. A typical bus interface allows the microprocessor to communicate with other devices using a “request/response” protocol. Using a microprocessor for high-speed control or data processing in an SOC requires a more flexible communication model. The TIE language allows you to create direct interfaces to other SOC blocks through the use of TIE ports, as illustrated below.

As explained earlier, you can create additional registers using the TIE state construct. Each designer-defined state can also be "exported" outside of the Xtensa LX processor using the export keyword in the state declaration. Exporting the state creates a new, dedicated output port. This output port can then be connected to other components on the SOC that need quick access to the value of the exported state.

The TIE language also supports an `import_wire` construct that defines a new input port to the Xtensa LX processor. This input port directly transports values into the processor and these values can be directly used by designer-defined TIE instructions. An `import_wire` is declared and used as follows:

```
import_wire EXT_STATUS 4

operation READ_EXTSTAT {out AR stat} {in EXT_STATUS} {
     assign stat = EXT_STATUS;
}
```

The `import_wire` declaration above creates a 4-bit input bus named `EXT_STATUS` at the boundary of the Xtensa core that can be connected to any other SOC block in the system. This bus is connected to the SOC block that provides this information to the Xtensa LX processor. Inside the processor, `EXT_STATUS` can be used directly as an input operand to any designer-defined TIE instruction as illustrated with the instruction `READ_EXTSTAT`. This instruction reads the value of `EXT_STATUS` and assigns it to an Xtensa LX core `AR` register. Subsequently, any of the Xtensa core instructions can be used to process this value. Note that the imported state `EXT_STATUS` can also be directly used as an input operand in a computation without first moving it into an intermediate state register or a register file.

Reads from an `import_wire` should not have any effect on the other system blocks, that is, this wire (or bus) can be read multiple times by instructions executed by the Xtensa LX processor without creating side effects. The Xtensa LX processor generates no output that indicates to external logic when the imported wire (or bus) is being read. The `import_wire` construct is often used to poll the status of a device or of another processor on the SOC.

A third type of designer-defined port, in addition to the exported state and imported wire, is the queue construct, which automatically masks the speculative nature of the `import_wire` construct. A queue is commonly used in hardware design to communicate between two modules. The module that generates the data *pushes* data onto the queue, and the module that consumes the data *pops* data from the queue at a later time when it is ready to process that data. The TIE language makes it easy to create new interfaces that communicate with input and output queues. The following example describes a TIE instruction that reads from an input queue, adds a value to it, and writes the result to an output queue.

```
queue InQ 32 in
queue OutQ 32 out

operation MQ {in AR val} {in InQ, out OutQ} {
     assign OutQ = InQ + val;
}
```

The first two statements declare an input queue named `InQ` that is 32 bits wide and an output queue named `OutQ` that is also 32 bits wide. Input queues can be used like any other input operand to an instruction and output queues can be assigned values just like any other output operand.

The TIE compiler translates the queue declaration above into a set of interface signals at the boundary of the Xtensa LX processor. The input queue declaration translates to an input signal named `TIE_InQ_Empty`, a 32-bit input bus named `TIE_InQ`, and an output signal named `TIE_InQ_PopReq`. Note that the actual hardware queue is outside the Xtensa LX processor and these signals interface to the “read” end of the hardware queue.

In the simplest case, the Xtensa LX processor asserts the `TIE_InQ_PopReq` following the execution of an `MQ` instruction, indicating its desire to read from the input queue. The data on the bus `TIE_InQ` is read, assuming that there is data in the queue to be read (`TIE_InQ_Empty` is false). If the input queue is asserting its empty signal (indicating that the input queue is empty), the `MQ` instruction will stall until the queue is no longer empty.

Complications to this simple case arise when the `MQ` instruction is killed in the processor’s pipeline (by an exception, for example) because processor reads from TIE queues and TIE ports are speculative. If the `MQ` instruction reaches the pipeline’s E stage before it’s killed, the processor will initiate the input-queue read operation. If the `MQ` instruction is subsequently killed, then the data obtained from the subsequently completed input-queue operation will be stored in a buffer inside the Xtensa LX processor and is used by the next `MQ` instruction that reads from the queue.

The TIE compiler translates the output queue declaration into an input signal named `TIE_OutQ_Full`, a 32-bit output bus named `TIE_QutQ`, and an output signal named `TIE_OutQ_PushReq`. When the `MQ` instruction executes, it makes the result of the computation available on the `TIE_OutQ` data bus, and asserts `TIE_OutQ_PushReq`. If the `TIE_OutQ_Full` signal is deasserted, the data is assumed to have been written to the queue. On the other hand, the processor stalls if the output queue is full (the `TIE_OutQ_Full` signal is true). Output queues are only written after the instruction has gone past the commit stage of the processor pipeline, so queue-write operations are never killed.

TIE ports and queues allow the hardware designer to create custom interfaces between modules, in much the same way that such interfaces are created during the design of logic blocks using RTL descriptions. Thus the Xtensa LX processor is not limited to the fixed interfaces, interface protocols, and limited bandwidth of traditional processor bus interfaces.

The TIE compiler automatically generates all of the logic necessary to generate the control signals that interface to the queue, the logic to detect the stalls, and the buffering of the speculative reads in the case of an input queue, based on the simple declaration of an input or output queue. TIE ports and queues can be simulated using the Xtensa Instruction-set simulator (ISS) in the XTMP simulation environment.

16. System Design Using TIE Ports and TIE Queues

Most TIE constructs result in the generation of data path and control logic internal to the Xtensa LX microprocessor. All the interface signals to and from these TIE modules are already connected to the appropriate interfaces of the baseline Xtensa LX microprocessor core. However, TIE ports and TIE queues result in new interfaces at the boundary of the Xtensa LX microprocessor. The designer is then responsible for connecting these interface signals to appropriate modules in their design.

This chapter provides general information on connecting these interfaces correctly. Timing diagrams are also provided to help the system designer understand the behavior of these interface signals.

16.1 TIE Ports: State Export and Import Wires

Designers can create their own top-level Xtensa LX processor interfaces with the TIE state export and import wire constructs. These interfaces can be used for many purposes, including multi-processor synchronization, device control, and status reporting.

16.1.1 TIE State Export

The TIE state export construct allows designer-defined state to be visible to hardware external to the Xtensa LX processor, via top-level pins. The exporting of designer-defined state may be a useful system-design tool for efficiently exposing status and control information between multiple Xtensa LX processors and devices.

If a state is exported, the architectural copy (committed state) is visible to the top-level pin named `TIE_<`*`name`*`>`, as shown in Figure 39.

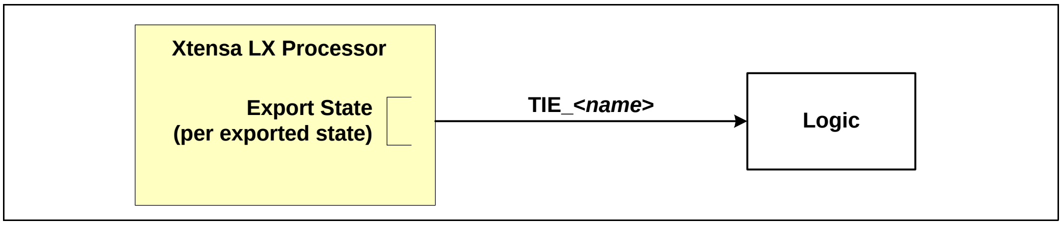


Figure 39. Exported State Block Diagram

16.1.2 TIE Import Wires

Import wires allow designer-defined TIE instructions to read inputs from designer-defined, top-level pins of the Xtensa LX processor. These interfaces can be connected to external hardware, including other Xtensa LX processors and hardware devices. For example, an import wire can be used to read the exported state of another Xtensa LX processor.

Import wires are always registered in the Xtensa LX processor before they become available for use in instruction semantics. Registering of import wires allows external logic to have late timing on the input wire, while giving instruction semantics an early timing.

Import wires are available for use in the E-stage of TIE instructions. Thus, such reads are speculative, because the instruction may be killed before it reaches the commit stage of the processor pipeline. If the instruction is killed, the read value is simply discarded. It is also assumed that the value on the import wire is always available to be read; there is no way to stall the processor on the read of an import wire. No “handshake” is involved in the reading of an import wire. Thus, the external logic does not receive any acknowledgement of when the input wire was read.

16.1.3 A Simple Multiple Processor System Communicating via TIE Ports

In the example system block diagram below, a master processor called Xtensa LX processor 1 can force a slave processor, called Xtensa LX processor 3, into a stalled state via the RunStall port with an export state, TIE_StallProc. The master processor can then cause another processor, Xtensa LX processor 2 (which is serving as a DMA engine) to initialize the instruction memory of processor 3.

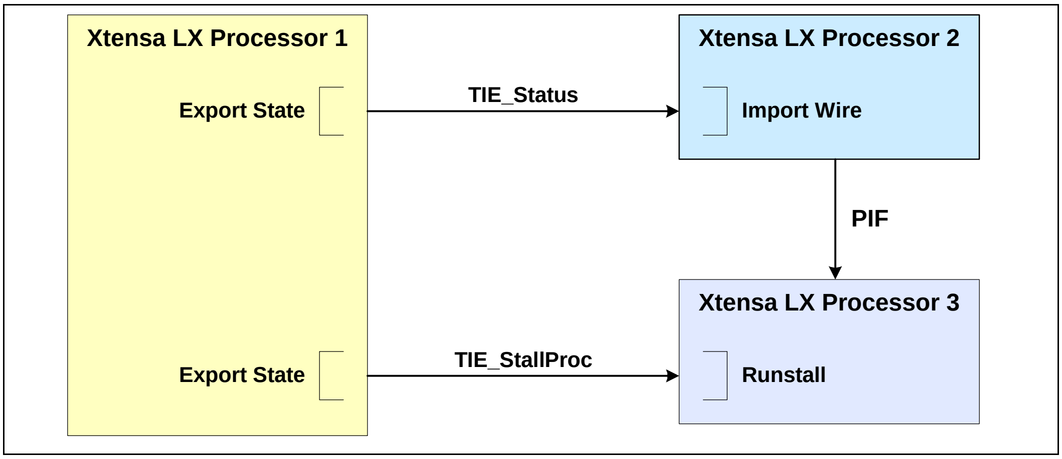


Figure 40. Example System Connected through Export State and Import Wires

Many other systems and uses of exported states and import wires are possible.

16.2 TIE Queues

TIE queues allow Xtensa LX processors to directly interface to external queues that have standard push- and pop-type interfaces and with full-and-empty flow control. Designer-defined instructions can push data from the processor onto external queues, pop data from external queues into processor state registers or register files, or use the information from input queues directly in computations.

The queue interface should be connected to synchronous, First-In, First-Out (FIFO) type devices only. The Xtensa LX queue interface is synchronous in the sense that the data transfer between the Xtensa LX processor and the external queue occurs on the rising edge of the clock signal.

Figure 41 shows the interfaces that are available for TIE queues.

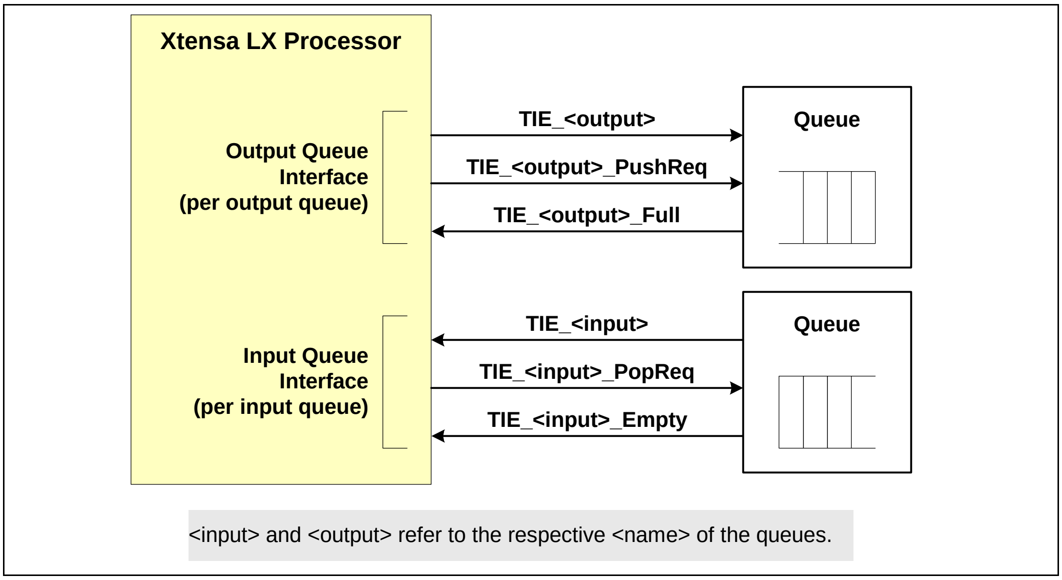

Xtensa LX Processor
Output Queue Interface (per output queue)
Input Queue Interface (per input queue)
TIE_<output>
TIE_<output>_PushReq
TIE_<output>_Full
Queue
TIE_<input>
TIE_<input>_PopReq
TIE_<input>_Empty
Queue
<input> and <output> refer to the respective <name> of the queues.

Figure 41. Interface Signals for TIE Queues

16.2.1 TIE Input Queues

A TIE input queue is used to read data from an external queue. Queue data is available for use in instruction semantics in the processor pipeline's M stage. The queue data is registered within the processor and is therefore read at least one cycle before the data is needed by an instruction in the M stage.

When a TIE input queue is defined, a `PopReq`-and-`Empty` interface is automatically created along with the queue's data interface. The Xtensa LX processor asserts the associated `PopReq` (pop request or read enable) signal whenever it wants to read data from an input queue. If `PopReq` is asserted and `Empty` is not, the Xtensa LX processor latches the data on the data interface and the read operation is completed. The external queue is expected to present valid data on the data interface whenever the empty signal is not asserted.

If the `Empty` signal is also asserted in a cycle when the Xtensa LX processor asserts `PopReq`, no data transfer takes place. In this situation, the Xtensa LX processor may keep the `PopReq` asserted in the next cycle, or it may remove the request. The pop request may be removed for a variety of reasons, such as the occurrence of an interrupt. Thus, the TIE input queue's data-transfer protocol is a cycle-by-cycle based protocol, with each cycle being independent of the previous cycle(s).

If the queue's `Empty` signal is asserted when the Xtensa LX processor wants to read data from an input queue and instruction execution cannot continue without the data, the processor's pipeline will stall and wait until the `Empty` signal is no longer asserted. This behavior is called a *blocking-queue* behavior. Note that the Xtensa LX processor may keep its pop-request signal asserted when the external queue is empty. The queue design should be such that this does not cause the queue to underflow; instead, it must ignore the processor's pop request when the queue is empty.

As with many read operations, the Xtensa LX processor can perform speculative reads on TIE input queues. The external queue is read before an instruction's commit stage, so there is a possibility that the instruction using the popped queue data may not commit due to a control-flow change or an exception. The processor handles the speculative nature of external queue reads by internally buffering all queue reads.

TIE queue reads are always speculative, because an external queue may be popped and the instruction that popped the queue may not commit its write to any architectural state. The processor buffers the queue data internally until the next queue-read instruction executes. That instruction will read the value saved in the processor's internal buffer rather than a new value from the external queue. There can an arbitrary amount of time between when an external queue is popped and when the popped data is consumed by an instruction executing in the processor.

The TIE input queue interface serves as a decoupling mechanism between the external queue and the Xtensa LX processor pipeline. It provides sufficient buffering to allow data to be popped from the external queue every cycle to allow sustained, high-bandwidth data transfers from the external queue.

The following TIE code example and subsequent timing diagrams illustrate the various aspects of an input queue interface. The example instantiates an input queue named `QIN`, an exported state named `LAST_READ`, and an instruction that reads the `QIN` input queue and writes that value to the `LAST_READ` state.

```
queue QIN 32 in
state LAST_READ 32 32’b0 add_read_write export
operation READQ {} {out LAST_READ, in QIN} [
     assign LAST_READ = QIN;
}
schedule READQ {READQ} { def LAST_READ 2; }
```

16.2.2 TIE Output Queues

The Xtensa LX processor uses TIE output queues to write data to external queues. Instructions that write to an output queue must have the data available in their M stage. Data written to an output queue is always internally registered before reaching the Xtensa LX processor’s output pin, maximizing the amount of time within a clock cycle that logic external to the processor has to operate on the output-queue data. This means that the data is first available to the external queue in the instruction’s W stage. If the instruction writing to the output queue does not commit, then the processor will kill the output queue write before it attempts to write to the external queue. Thus, the Xtensa LX processor never performs speculative writes to external output queues.

Defining a TIE output queue automatically creates a `PushReq`-and-`Full` interface along with the data interface to the external queue. The `PushReq` (push-request or write-enable) signal is asserted whenever the Xtensa LX processor wishes to write data to the output queue. When the processor asserts `PushReq`, it also presents valid write data on the queue’s data interface. If `PushReq` is asserted and `Full` is not, the Xtensa LX processor assumes that the external queue has latched the data and that the write transaction is complete.

If the `Full` signal is also asserted in a cycle when the Xtensa LX processor asserts `PushReq`, no data transfer takes place. In this situation, the Xtensa LX processor may continue to assert `PushReq` during the next cycle, or it may remove the request. Thus, the output-queue transfer protocol is cycle-based, like the input-queue transfer protocol. Each transfer cycle is independent of the previous cycle(s). Note that the Xtensa LX processor may keep its push-request signal asserted when the external queue is full. The external queue should be designed so that this state does not cause the external queue to overflow; the external queue must ignore the push request when the queue is full.

The Xtensa LX processor is able to commit and buffer two queue-write instructions made to a full queue without stalling the processor. A third queue write to a full queue will stall the processor. This buffering ensures that instructions that write to a TIE queue can be issued in a fully pipelined manner. The buffered data is subsequently written to the external queue when it is no longer full.

If the Xtensa LX processor is interrupted during an output-queue access, all committed output-queue writes will still be performed without data loss.

The following TIE code illustrates the previous points. It instantiates an output queue called `QOUT`, and an instruction, `WRITEQ`, that writes to the queue.

```
queue QOUT 32 out
operation WRITEQ {in AR art} {out QOUT} [
     assign QOUT = art;
}
```

16.2.3 TIE Queue Interface Recommendations

The TIE input- and output-queue interfaces have been designed to mate directly to synchronous FIFO devices. The TIE queue interfaces are synchronous in that all data transfers between the Xtensa LX processor and the external queues occur on the rising edge of the clock signal. There should be no combinational paths from `PopReq` to `Empty` in input-queue interface logic and from `PushReq` to `Full` in output-queue interface logic. Thus, if an input queue becomes empty as a result of a pop request from the Xtensa LX processor, the queue should assert its `Empty` signal in the next clock cycle, following the clock edge marking the data transfer. Similarly, if an output queue becomes full as a result of a push request, it should assert its `Full` signal in the next clock cycle after the clock edge, when it accepts the data from the processor.

The Xtensa LX processor may assert its pop request signal when an input queue is empty, or it may assert its push request signal when an output queue is full. External queue designs must protect the queues against input-queue underflows and output-queue overflows by ignoring the processor’s requests when an input queue is empty or when an output queue is full.

The Xtensa LX processor should be the only device connected to the read port of an input queue or the write port of an output queue. TIE queue interfaces are not designed to connect to external queues that are shared between multiple clients. During normal queue operation, the status of an input queue must change from “not empty” to “empty” only in response to a pop request from the Xtensa LX processor. Similarly, the status of an output queue must change from “not full” to “full” only in response to a push request from the Xtensa LX processor. The only situation where it may be appropriate to violate this particular restriction (for both input and output queues) is during a system flush. Such a flush might occur, for example, before data processing begins on a new stream of data.

System designers must ensure that the Xtensa LX TIE queue interfaces are used in a manner consistent with the recommendations outlined in this section. Not doing so may result in unexpected or incorrect queue behavior.

16.2.4 TIE Queues Compared to the XLMI Port

External queues can also be connected to the Xtensa LX through its XLMI port. To choose between the XLMI port and TIE queues to connect to external queues, consider the following points (in addition to other system constraints):

- FIFO queues connected to the XLMI port are accessed via memory-mapped load and store instructions, which can be either standard Xtensa LX instructions or designer-defined TIE instructions. External queues attached to the processor through TIE queues *must* employ designer-defined push and pop instructions, because the TIE queue interfaces do not exist within the processor's address space. The designer should consider whether memory-mapped or TIE queue push and pop operations are the more appropriate usage model. The processor can transfer data to or from only one XLMI-attached queue during any given cycle (because there's only one XLMI port), but it can transfer data to and from several TIE queues during each clock cycle. XLMI-attached queues require no additional I/O lines other than the XLMI port but each additional TIE queue interface adds I/O lines to the processor. For some system designs, the interface choice won't matter.
- An Xtensa LX processor configuration cannot have more than one XLMI port, but it can have any number of TIE queues. If a FIFO queue is attached to the XLMI port, it must share that port with any other XLMI-attached devices, which may result in bandwidth issues. TIE queue interfaces are not shared, and therefore an external queue attached to a TIE queue interface receives the full bandwidth of that interface.
- If the processor tries to read from an input queue when it's empty, an XLMI-attached FIFO queue will stall the I/O operation. The stalled I/O read will stall the processor's pipeline as well. This stalled state is not interruptible. TIE queues can stall the processor's pipeline but interrupts can still occur when a TIE queue is stalled.
- If the processor executes a store to the XLMI port immediately followed by a load from an XLMI-attached FIFO queue, the store is buffered and the load occurs first. If this load causes an I/O stall because the addressed FIFO queue is empty, a resource deadlock will freeze the processor's pipeline. The XLMI load cannot complete because the FIFO queue is empty and the store to the FIFO queue cannot take place because the pipeline is stalled.
- All XLMI-attached devices including FIFO queues must handle speculative reads over the XLMI port. To do this, XLMI-attached devices must observe and use the processor control signals that indicate whether a read has committed or flushed. The Xtensa LX processor automatically handles speculative reads for TIE input-queues.
- For Xtensa LX processor configurations with a PIF, the XLMI port width will be the same width as the PIF (32, 64, or 128 bits wide). For Xtensa LX processor configurations without a PIF, the XLMI port can be 32, 64, or 128 bits wide. TIE queues can be of any width from 1 to 1024 bits and different queues on the same processor configuration can have different widths.

17. Xtensa LX EDA Development Environment and Hardware Implementation

This chapter describes the EDA development environment for implementing SOCs using the Xtensa LX processor.

17.1 Design Environment

The Xtensa LX processor core ships with a complete set of implementation scripts that drive the various synthesis, verification, layout, and test tools through all the steps necessary to ensure a design with high quality, high performance, and fast time-to-market.

The Xtensa LX configuration environment directly supports the choice of design targets, EDA tools, and flows used in a particular project. Along with the Xtensa LX processor itself, the configuration system automatically generates optimized scripts based on the configuration choices. These scripts can be used as provided or as starting-point templates for script customization to achieve better quality results. In addition, the configuration system is fully automated, permitting easy design customization as well as design exploration. These characteristics significantly improve design turn-around time.

The designer begins by selecting configuration parameters in the Xtensa configuration editor in Xtensa Xplorer. These parameters include information about the design's performance, area and power targets, and the development team's EDA tools and flows. After the configuration data is submitted, the processor generator creates the appropriate scripts for the selected EDA tools. These scripts are tuned according to the specified design targets and selected hardware configuration options.

The next step is for the designer to download the automatically generated files and begin the implementation and verification process. After customizing a simple configuration file, the designer can simulate the processor RTL; synthesize it; simulate the resulting post-synthesis, gate-level netlist; and perform physical design and timing verification. These automated steps give the designer a quick and accurate view of the timing, power, and area achievable in the designer's tool environment.

The configuration file also allows the designer to explore the implementation space and make trade-offs by altering the set-up variables. A makefile helps automate design exploration. The net result is that within few hours or a few days, depending on the design size and constraints, the designer can develop highly accurate timing, area, and power numbers in the chosen IC-implementation technology. After the system design is finalized and all Xtensa LX configuration options and TIE extensions have been selected, the hardware design team downloads a production version of the processor RTL from the Xtensa LX Processor Generator and proceeds with the final chip assembly, integration, and verification with full confidence.

Figure 42 illustrates the Tensilica processor implementation flows. There are two main flows to choose from: the physical-synthesis flow and the more traditional place-and-route (P&R) flow. The design team should choose the implementation methods based on their design needs and EDA tool availability.

The physical-synthesis flow effectively solves the timing-correlation problem between logic synthesis and the placement-and-routing step that typically appears for the traditional flow. Therefore, the physical-synthesis flow greatly reduces or removes the need for repetitive iteration between logic and physical design. This flow is most suitable for large designs with high-performance targets, but it requires use of a physical-synthesis tool such as Synopsys's Physical Compiler or Cadence's PKS.

The traditional P&R flow is more suitable if design goals emphasize area optimization and performance is not as high a priority. For the traditional flow, we advise the designer to create a custom wire-load model to aid the first attempt at placement and routing and to use this wire-load model in subsequent logic-synthesis runs. A custom wire-load model helps to correlate timing between logic and physical design and helps reduce the iteration between them. Wire-load models are especially needed when the chip design is larger than 50k gates using nanometer IC-fabrication technology, and nearly any processor-based IC design using advanced IC lithography will be far larger than 50k gates. An additional power-optimization step will also be useful if the design's power consumption is critical.

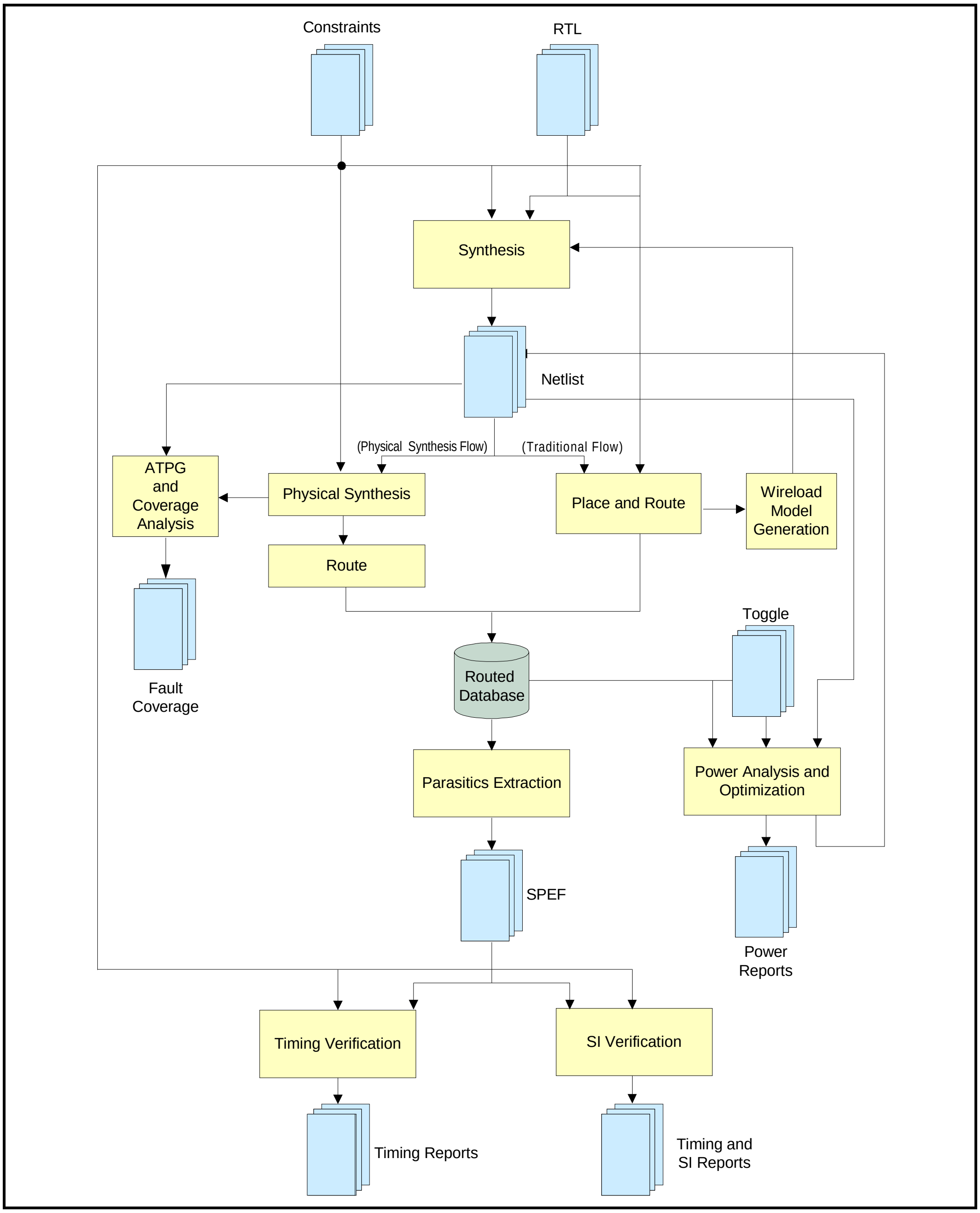


Figure 42. Xtensa LX Processor Implementation Flows

17.2 Logic Synthesis

Xtensa LX processor RTL is synthesized using Synopsys' Design Compiler to generate a timing- or area-optimized gate-level netlist. Tensilica provides a set of Design Compiler reference scripts that are optimized for the Xtensa LX processor core. The Xtensa LX scripting infrastructure allows designers to specify any standard-cell library, design constraints, and optimization switches.

The default I/O constraints for the Xtensa LX processor are generated automatically. The PIF timing defaults are based on AC timing specifications. Memory-port timing defaults are derived using industry-standard memory compilers. For optimum results, the designer should adjust the provided default time budgets to reflect specific memory and system requirements.

The generated scripts implement a bottom-up compilation strategy with multiple passes at the processor level. Optimization switches allow developers to perform speed, area, and turnaround-time trade-offs. The scripts can automatically identify and implement register rebalancing for TIE semantic modules (designer-defined processor execution units) that require multiple pipeline stages.

The scripts use a path-grouping technique to ensure that flop-to-flop paths are optimized independently of I/O paths within the processor RTL. This technique produces optimal solutions for internal paths in the Xtensa LX processor, even in presence of approximate timing budgets that might evolve during the design process to suit changing system requirements.

The Xtensa LX processor design aggressively gates the clock to minimize dynamic operating power. During synthesis, the scripts model buffering delays from the output of a gated clock to the registers by specifying maximum delay constraints to the enable pins of the gated clock.

The scripts also optionally support single-pass, design-for-testability (DFT) scan synthesis.

17.3 Physical Synthesis and Routing

For 180nm and smaller IC-fabrication technologies, wire delay is a significant percentage of the total path delay. Use of fanout-based wire-load models leads to timing-convergence issues and potentially suboptimal results. The EDA industry has addressed this issue by combining the logic-synthesis and placement steps together in physical synthesis.

Tensilica's Processor Generator creates scripts for Synopsys' Physical Compiler (PC) physical-synthesis tool. The generated scripting infrastructure includes optimization switches to manage timing, congestion, and cell area when using PC. The generated

PC scripts also perform scan-chain deletion and restitching to avoid scan-path wiring jumbles resulting from placement. Scripts are provided to complete clock-tree synthesis, timing optimization, and routing using either Cadence's Silicon Ensemble or Synopsys' Apollo tools. The processor generator also creates a script for a Tensilica-proprietary tool (the XT floorplan initializer) that creates the initial floorplan required by PC. In addition, the processor generator creates scripts for Cadence's Physically Knowledgeable Synthesis (PKS) product.

17.4 Placement and Routing

The P&R step takes a gate-level netlist and performs floor planning, placement, clock-tree synthesis, timing optimization, and routing to produce a finished design. The processor generator creates P&R scripts for both Synopsys' Apollo and Cadence's Silicon Ensemble tool suites.

An option is provided to create a custom wire-load model based on the routed design, which is typically used for the first pass P&R. For subsequent design iterations, the generated custom wire-load model can be used for logic synthesis to improve the timing correlation between logic synthesis and the P&R step.

17.5 Parasitic Extraction

Parasitic extraction takes a routed design database, performs 3D RC extraction, and produces parasitic data in a SPEF file format. The supported vendor tool is Cadence's Fire&Ice.

Depending on the designer's verification needs, an option to extract cross-coupling data and perform signal-integrity analysis is provided.

17.6 Timing Verification

The timing-verification step takes a routed netlist, timing constraints, and the extracted SPEF file and then performs min-max timing analysis and generates delay and timing report files. The supported vendor tool is Synopsys' PrimeTime.

17.7 Signal Analysis, Noise-Failure Repair, and Sign-off

Cadence's Celtic signal integrity (SI) analysis and repair scripts are provided to generate SI ECO fix information for cross-talk noise glitches and to perform SI-aware timing signoff. The SI analysis and repair flow takes generic form SPEF with coupled-RC data generated by Fire&Ice, iterates between Celtic and PrimeTime, and creates converged timing windows and incremental SDF for the final static timing analysis. Celtic also generates a new Verilog or DEF ECO file for the P&R tools, depending on which file was provided to the Celtic run deck to fix noise failures.

17.8 Power Characterization and Optimization

Tensilica provides scripts to perform gate-level power analysis on the Xtensa LX core. The same diagnostics used to perform functional verification also generate toggle activity for the core. This toggle file, in conjunction with a gate-level netlist and loading information, is used to analyze a processor design's total power consumption. These power numbers are reported in terms of gate, net, and leakage components. Other provided scripts perform power optimization on a netlist, which is achieved primarily by downsizing gates along non-critical paths. Both power analysis and optimization are performed within Design Compiler and require a Power Compiler license.

of the processor core). Moreover, Tensilica continuously runs co-simulation regression tests to correlate RTL and ISS (instruction-set simulator) simulation results. This exhaustive testing produces a processor core that is highly verified and robust.

17.10.2 Designer-Defined TIE Verification

Correct-by-construction verification of designer-defined TIE implementations is performed by using test vectors written into the designer-written TIE source code to generate architectural verification diagnostics. The processor generator also generates additional micro-architecture diagnostics to verify the processor-core-to-TIE interface. These steps ensure that the designer-defined TIE instructions are implemented in the generated RTL *exactly as the designer described them*.

The Xtensa LX verification environment also supports formal verification (using Incisive Conformal) to ensure that the designer-defined TIE references and semantics are logically equivalent.

17.10.3 SOC Pre-silicon Verification

A system-on-a-chip (SOC) design may consist of various intellectual property (IP) blocks integrated with one or several Xtensa LX processors. Pre-silicon SOC verification at the top level is accomplished using various methods. These methods include the Xtensa LX processor's instruction set simulator (ISS) used with the Xtensa Modeling Protocol (XTMP) API environment, RTL simulation, FPGA-based emulation using the XT2000 Emulation Kit, and hardware/software co-verification using third party tools such as Mentor's Seamless with Xtensa LX processor's ISS and co-simulation model. It is left up to the SOC design team to individually verify that IP blocks other than the Xtensa LX processor(s) function as specified.

Tensilica also provides VERA monitors to check for violations of interface protocol during simulation. Verifying SOC connectivity to the Xtensa LX processor(s) is an important phase in pre-silicon SOC verification. These monitors aid the verification and debugging of logic that interfaces to the Xtensa LX Processor(s), either through the PIF or through the local memory interfaces or the XLMI port. The monitors also provide immediate verification feedback if the system violates the interface specifications. Consequently, the verification engineer need not write diagnostics for the Xtensa LX interfaces and can instead spend more time on system-level testing.

17.10.4 SOC Post-silicon Verification

The Xtensa LX core is a fully synchronous design that is completely compatible with JTAG full-scan and partial-scan manufacturing test methodologies. Tensilica uses the check_test feature of Synopsys' Design Compiler to ensure that the resulting processor design complies with JTAG scan requirements during the implementation process. This feature is very conservative on the types of logic it considers testable. Consequently, the scan-insertion and ATPG flow is fully compatible with all of the industry-standard tools currently in use. Tensilica has exercised and tested these flows during its own silicon prototype verification efforts.

The Xtensa LX processor implements the special Xtensa ISA instructions that directly read and write the cache data and tag RAM arrays. These instructions can be used to implement manufacturing memory tests and a power-on self-test (POST).

18. Xtensa LX System Designer's Guide

The Xtensa LX Processor Generator creates processor RTL in Verilog or VHDL, along with test benches and monitors for hardware simulation. However, the Xtensa LX processor is more than a hardware design. The processor generator also creates a complete software design environment tailored to the configured Xtensa LX processor. This environment includes tools for writing applications, for debugging and profiling programs, and for managing processor configurations. Table 91 summarizes the development tools provided with the Xtensa LX processor RTL.

The tools include the Xtensa instruction set simulator (ISS) and several processor simulation models. System designers may use these models to simulate their entire system at different levels of precision.

Table 91. Summary of Xtensa LX Development Tools

Tool	Description
Application development:	
`xt-xcc`	Xtensa C Compiler
`xt-xc++`	Xtensa C++ Compiler
`xt-ld`	GNU Linker
`xt-as`	GNU Assembler
Xplorer	Xtensa Xplorer Development Environment - Standard Edition
Application simulation:	
Xplorer	Xtensa Xplorer Development Environment - Standard Edition
`xt-run`	Xtensa Instruction Set Simulator
`xt-gdb`	GNU Debugger
`xt-gprof`	GNU Profiler
Configuration management:	
Xplorer	Xtensa Xplorer Development Environment - Processor Developer's Edition
System simulation models:	
XTMP API	Xtensa Modeling Protocol
CVE CSM	Xtensa Co-Simulation Model for Mentor Graphics Seamless

18.1 System Simulation Glossary

Note: The following definitions are generic in nature and are not specific to any solutions provided by Tensilica.

- *Hardware (HDL) Simulator:*
  A simulator for a hardware design language, such as VHDL or Verilog. HDL simulation is slow but very accurate.
- *Instruction Set Simulator (ISS):*
  A high-performance, software-only processor model and simulator. An ISS runs programs compiled for the actual processor, executes all instructions, and produces the same results as the processor. ISS models are very fast, but may not produce cycle-count timing results as accurately as HDL simulations. Some ISS models simulate only at the instruction level (they do not model the pipeline) and provide only approximate cycle counts.
- *Processor Co-Simulation Model (CSM):*
  A model of a processor's external interface that performs all of the processor's bus transactions. Some CSMs are simple, script-driven models. Others are controlled by an ISS and can run any processor program.
- *Hardware/Software Co-Simulation:*
  An environment in which a software model of a processor drives a CSM, which connects to other HDL components in a hardware simulator. Co-simulation can be much faster than simulating the entire system with an HDL simulator. Also known as hardware/software co-verification.
- *Bus transaction-level model:*
  A model of a processor bus in which all operations are represented by high-level transactions, rather than values on bus signals.
- *Bus signal-level model:*
  A model of a processor bus that accurately models all of the bus-interface signals.

18.2 Xtensa LX System Models

Tensilica provides several models for the Xtensa LX processor that can be used for system simulation. These simulation models range from high-level software simulations of the Xtensa LX processor to the synthesizable Verilog or VHDL that defines the processor. While it is possible to simulate a large multiple-processor system entirely in Verilog or VHDL, the simulation would run very slowly. Developing and debugging software is also harder using hardware simulation. Table 92 lists system-modeling alternatives for designs based on Xtensa LX processors and the relative speeds of these alternatives.

Table 92. Xtensa LX System Modeling Environments

Simulation speed (cycles/second)	Modeling Tool	Benefits
200K to 400K [*1, *2]	ISS-XTMP API	▪ Multiple processor system modeling ▪ Cycle accurate pipeline ▪ System architecture exploration
4K to 50K [*1, *2]	ISS-CSM for Mentor Graphics Seamless	▪ System hardware modeling ▪ Cycle accurate pipeline ▪ Vary performance vs. accuracy
50 to 200 [*1, *2]	Behavioral & RTL hardware simulation	▪ Functional verification ▪ Logically equivalent to final hardware ▪ High visibility and accuracy

[*1] Simulation performance is an estimate, and is dependant on the host computer. These estimates are based on a 2-GHz host PC running Linux.

[*2] Simulation speed for a single Xtensa LX processor and memory subsystem.

Reviewed below are two different system-simulation models for the Xtensa LX processor. Both are based on the *Xtensa Instruction Set Simulator*. The Xtensa ISS is a fast, cycle-accurate model and processor simulator. By itself, the ISS allows developers to run and debug Xtensa LX processor programs. Two simulation packages add interfaces and libraries to the ISS to enable a designer to simulate a full system:

- *Xtensa Modeling Protocol (XTMP):*
  A programming interface (API) and library for the Xtensa ISS. XTMP allows developers to write their own C or C++ models of components that connect to Xtensa LX processors. XTMP provides a very efficient simulation environment, suitable for multi-processor system simulations.
- *Xtensa LX Co-Simulation Model (CSM) for Mentor Graphics' Seamless:*
  A *co-simulation* model for the Mentor Graphics' Seamless Co-verification Environment (CVE). This package adds a signal interface to the Xtensa LX XTMP model. It drives Xtensa LX I/O signals in a hardware simulation and can connect to other RTL system components.

Note: The *XTMP API* and the *Xtensa LX CSM* are provided only to Tensilica licensees who have purchased the system simulation and multiprocessor support option for the Xtensa LX processor.

18.3 Cycle Accuracy

The Xtensa LX XTMP API, and CSM models are all built on the ISS for the Xtensa LX processor. The ISS is a cycle-accurate model and simulator for the Xtensa LX processor. It accurately models the processor pipeline and simulates the interaction of the several instructions that occupy the processor's pipeline at any time. The ISS models caches, pipeline stalls, and resource contention. Used with the XTMP API, the ISS can interact with cycle-accurate models of memory or other system devices to help predict the performance of an entire system.

In almost every case, the ISS reproduces the cycle timing of the Xtensa LX processor hardware. The only known causes of inaccuracy are pipeline stalls related to the store-tag buffer (in Xtensa LX configurations with a write-back data cache), and arbitration for the load/store unit between inbound-PIF and internal memory requests.

19. Xtensa Modeling Protocol (XTMP) Environment

Tensilica's advanced Instruction Set Simulator (ISS) and Xtensa Modeling Protocol (XTMP) environment perform fast and accurate simulation of SOC designs incorporating one or more processor cores. Running orders of magnitude faster than RTL simulators, the ISS/XTMP environment is a powerful tool for software development and SOC design.

The XTMP environment allows system designers to create customized, multi-threaded simulations to model more complicated systems. SOC designers can instantiate multiple Xtensa and Xtensa LX cores and use XTMP to connect these processor cores to designer-defined peripherals and interconnects. XTMP can be used for simulating homogeneous or heterogeneous multiple-processor subsystems as well as complex uni-processor architectures. Figure 43 shows a simple system simulation built with XTMP.

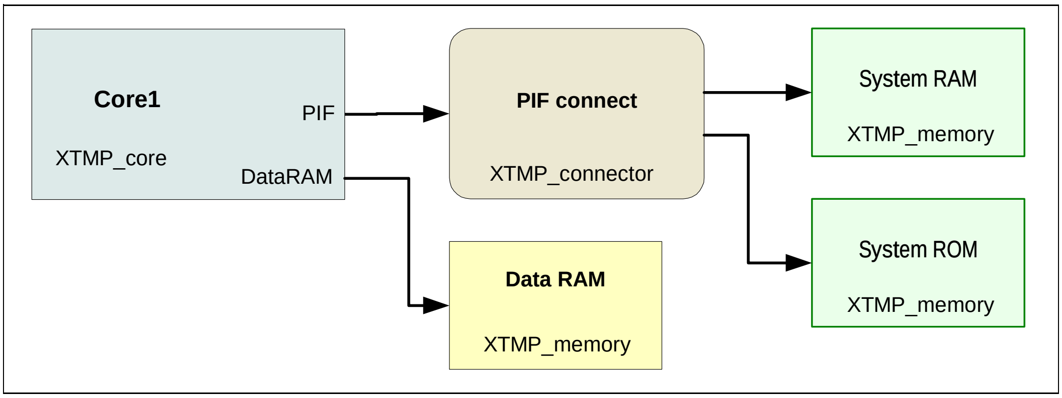


Figure 43. A Simple System Built with XTMP

The ISS/XTMP environment allows system designers to create, debug, profile, and verify their integrated SOC and software architecture early in the design process instead of waiting until the end of a design project to integrate the hardware and the software. Because XTMP device models are written in C and C++, the XTMP simulation runs faster than an HDL simulator running RTL models. XTMP device models are also much easier to modify and update than hardware prototypes.

A module written using the XTMP API is a *transaction-level* model. It represents bus and memory accesses using high-level transactions rather than by wiggling bus signals. A designer can write XTMP device and memory modules as cycle-accurate models or as untimed functional blocks.

19.1 Xtensa LX Co-Simulation Model for Mentor Graphics Seamless

The Xtensa LX Co-Simulation Model (CSM) for the Mentor Graphics® Seamless Co-Verification Environment (CVE) links a software XTMP model of the Xtensa LX processor to a hardware simulation of the rest of the system. Rather than driving signals with an RTL model of the Xtensa LX processor, the CSM uses the Xtensa instruction-set simulator (ISS) to drive hardware signals. This mixed-mode simulation allows designers to run and debug their system software and hardware much faster than a full RTL simulation.

Each co-simulation instance connects the Xtensa LX ISS to a cycle-accurate bus model of an Xtensa LX processor running in a Verilog or VHDL simulator. This bus model drives the Xtensa LX core’s output signals and samples the Xtensa LX processor inputs in the simulation. System developers can connect those signals to other HDL models in the hardware simulation. Figure 44 shows a system simulation built with both XTMP and CSM models.

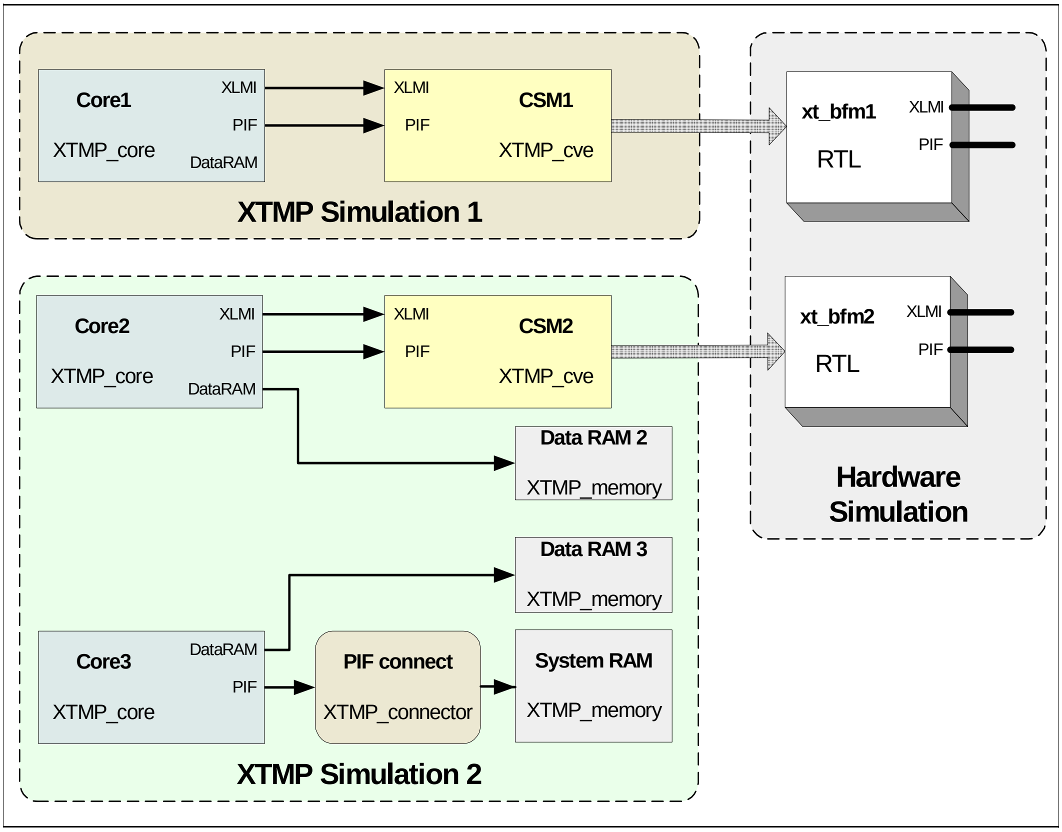


Figure 44. Mixed XTMP and RTL Simulation Using the Xtensa LX CSM

This figure shows a three-processor system simulation, in which two of the processor cores (core1 and core2) are connected to CSM models (CSM1 and CSM2), and one processor (core3) only connects to XTMP devices. In a CSM for Seamless simulation, each CSM processor model runs in a different instance of the ISS. The two ISS processes are completely separate. The CSM processor models can only communicate through signals and shared memory in the hardware simulation. However, each ISS instance can also contain other XTMP cores, devices and memories, which do not use CSM models.

The CSM does not drive all of the Xtensa LX I/O signals. The CSM models the following interfaces and ports:

- Xtensa LX system signals
- Interrupt signals
- PIF Master signals
- PIF Slave signals
- XLMI (Xtensa Local Memory Interface) Port signals
- TIE Input and Output Queues
- TIE state-export signals
- TIE import wires

The CSM controls all of the above ports in the simulation, if they exist in a processor configuration. The CSM drives output signals and samples input signals in the hardware simulation. Processor ports that are connected to a CSM model cannot be connected to other XTMP devices. The developer cannot specify, for example, that the CSM should model a processor's PIF interface but not its XTMP interface.

The CSM does not model:

- Instruction-RAM or -ROM interface signals
- Data-RAM or -ROM interface signals
- Instruction-cache interface signals
- Data-cache interface signals
- Debug and status signals

Local-memory RAM and ROM interfaces, which the CSM does not model, can be connected to XTMP memory models and devices. Cache interface ports and the debug and status signals do not have corresponding XTMP API calls, so these ports cannot be modeled through XTMP.

The speed of an Xtensa LX processor co-simulation ranges between that of a full RTL hardware simulation and a pure XTMP software simulation. The actual simulation performance depends on the number of processors in the simulated system and the amount of other logic in the hardware simulation. Developers can increase the simulation's speed (and reduce its accuracy) by optimizing memory accesses. An optimized memory access reads or writes directly from a software memory model and does not cause bus cycles in the hardware simulator. Refer to the *Mentor Graphics® Seamless User's and Reference Manual* (version 5.2) for a description of optimized memory accesses.

20. Hardware Emulation and the XT2000 Emulation Kit

The Xtensa Microprocessor Emulation Kit–XT2000 includes a complete emulation, evaluation, and development tool set for Tensilica's configurable Xtensa LX processor. These products enable the designer to evaluate various processor configuration options by developing and debugging software on prototype FPGA-based hardware very early in the design cycle. The XT2000, used in conjunction with the Xtensa LX Processor Generator, allows new Xtensa LX processors to be specified, implemented, and benchmarked in a day.

The XT2000-X emulation kit board uses a Xilinx FPGA for hardware emulation of Xtensa LX processor configurations. Tensilica’s processor generator automatically performs the synthesis, placement, and routing of the designer-configured processor core and an appropriate bus interface. The resulting FPGA configuration files are subsequently loaded by the designer into the XT2000’s FPGA from a host through a programming adapter cable.

There are three methods available for XT2000 board communication and control:

- RS-232 serial port
  The RS-232 serial port provides a communications link to a Sun or Windows NT workstation host for downloading and debugging software. The serial port works in concert with RedBoot, a resident debug monitor program.
- On-chip-debug (OCD) module
  For processor configurations with the optional OCD module, a JTAG port allows the processor to be viewed and controlled precisely via a Macraigor Systems Wiggler and the associated OCD daemon.
-

- Null-modem RS-232 cable to connect the XT2000 board to the serial port of a host PC or terminal
- XT2000 CDROM containing Solaris and Windows NT software tools, configuration files, and documentation

Figure 45 shows a block diagram of the XT2000-X emulation board.

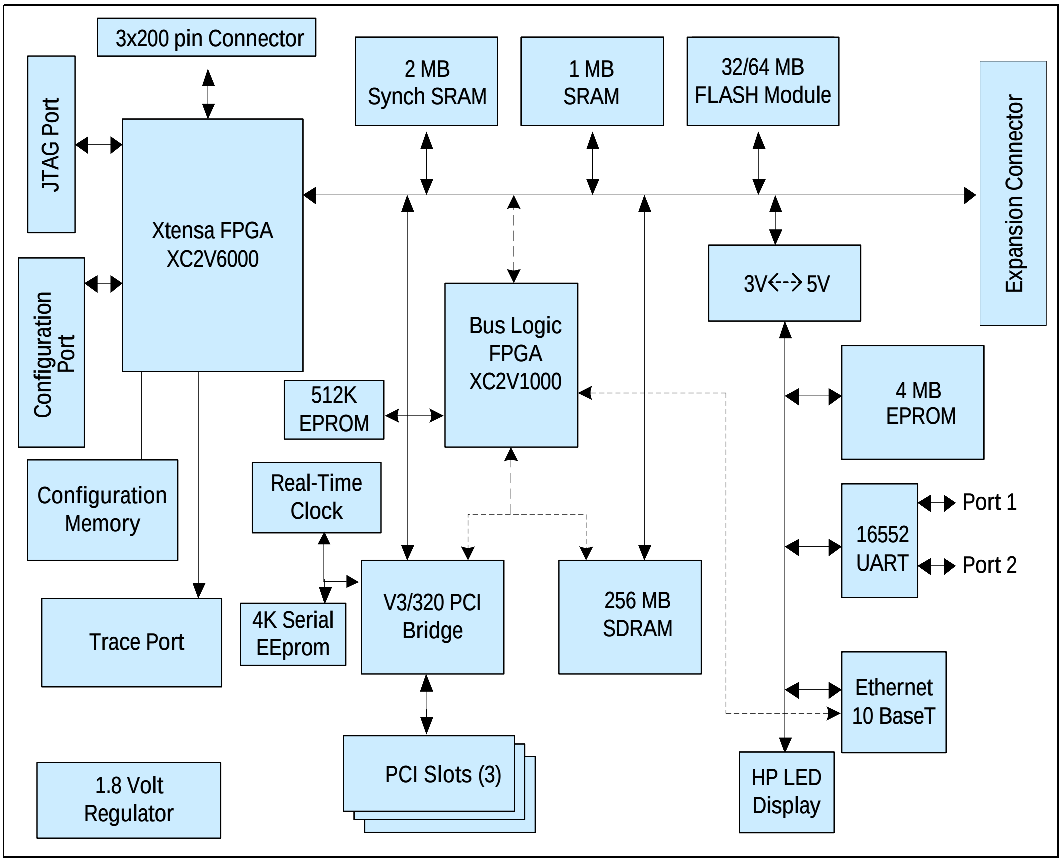


Figure 45. XT2000-X Emulation Board

20.2 Features

The XT2000 emulation board features include:

- A complete hardware emulation system for an Xtensa LX processor core
- Large FPGA to which you can download an Xtensa LX processor, bus logic, and peripherals
- 256Mbyte SDRAM
- 4Mbyte EPROM (two 16-bit devices)
- 512Kbyte PROM/EPROM (one 8-bit device)
- 2Mbyte synchronous SRAM (expandable to 8 Mbytes)
- 1Mbyte asynchronous SRAM
- 32Mbyte socketed Flash Module (socket expandable to 64 Mbytes)
- 4K-bit I^{2}C serial EEPROM (24C04A) for storing configuration information, Ethernet MAC address, etc.
- Three standard PCI slots (uses V3 V320USC PCI controller chip)
  - PCI I/O byte read operations are not supported
  - PCI-lock, atomic bus accesses are not supported
- 8-segment alphanumeric LED display
- Real-time clock (Dallas Semiconductor DS1672)
- 10BaseT (10Mbps) Ethernet (SONIC DP83934-CVUL)
- Two serial ports (PC16552 DUART)
- 120-pin expansion connector
- Debug support via either the Macraigor Systems Wiggler and the OCD daemon, the serial port and the RedBoot debug monitor, or the Ethernet port and the RedBoot debug monitor
- Real-time trace
- Power supply
- XLMI expansion headers

21. Using the XPRES Instruction-Set-Extension Compiler for Improved Design Productivity

The XPRES instruction-set-extension compiler creates a custom processor specifically designed to optimize the performance of a target C or C++ application or a set of target applications using automatically generated TIE extensions. Based on the target application(s), XPRES extends a base Xtensa LX processor with new instructions, operations, and register files specifically tailored to the application(s).

The new instructions, operations, and register files are described using TIE, so they are automatically recognized and exploited by the entire Xtensa LX software tool chain, including the C/C++ compiler. Thus, to take advantage of the custom processor, the designer need not make any changes to the target application's source code or perform any assembly language coding. The designer simply recompiles the target application(s), and the C/C++ compiler automatically exploits the new instructions, operations, and register files to accelerate application performance.

XPRES uses the Xtensa C/C++ compiler to analyze each target application and, based on this analysis and hardware-estimation techniques, XPRES automatically generates and evaluates thousands—or even millions—of possible custom processor extensions to accelerate the target application(s). For a range of hardware costs, XPRES then chooses the custom processor or processors that provide the maximum performance improvement.

For example, for a radix-4 FFT, XPRES automatically evaluates more than 34,000 custom extensions with estimated hardware costs ranging from 7800 gates to 128,000 gates. From these 34,000 possible extensions, XPRES selects 31 custom prototype processors that accelerate performance by factors of 1.12 to 11.3 compared to a base Xtensa LX processor. After XPRES has generated a set of custom prototype processors, the designer can simply choose the processor that most closely matches the design's performance and hardware-cost requirements. Because each custom processor is described using TIE, the designer can change the XPRES-generated TIE to add or remove operations, change register-file sizes, rename operations and register files, and otherwise refine the logic implementing the custom operations.

The XPRES Compiler offers the designer two methods for verifying the correctness of XPRES-generated operations. First, formal verification (using Cadence Incisive Conformal ASIC) can be used to ensure that the TIE reference and semantic created for each XPRES-generated operation are logically equivalent. Second, the XPRES Compiler creates a diagnostic that allows the designer to verify the architectural correctness of each XPRES-generated operation using instruction-set, RTL, or gate-level simulation.

XPRES uses several techniques to generate custom processor extensions, each of which has different trade-offs between potential performance improvement and hardware cost. These techniques include FLIX (Flexible Length Instruction Xtensions), vector (SIMD) operations, and operation fusion, which are discussed elsewhere in this data book.

FLIX allows one instruction to contain multiple independent operations. A FLIX-format instruction is partitioned into a number of slots, each of which may contain a set of operations. If XPRES designs the custom processor using FLIX, the Xtensa C/C++ (XCC) compiler will use software-pipelining and instruction-scheduling techniques to automatically pack multiple operations into a FLIX instruction, which can significantly increase performance compared to a single-instruction-issue Xtensa LX processor. However, designing a custom processor using FLIX increases gate count, because a FLIX instruction requires additional logic to issue and execute multiple independent operations in parallel.

The vector-operations extension technique increases performance by creating operations that work on more than one data element at a time. Vector operations are also referred to as Single Instruction Multiple Data, or SIMD, operations. A vector operation is characterized by the operation it performs on each data element, and by the number of data elements that it operates on in parallel.

For example, a four-wide vector integer addition operation sums two input vectors, each containing four integers, and produces a single result vector consisting of four integers. If XPRES designs a custom processor to use vector operations, the xt-xcc compiler will use automatic parallelization and vectorization techniques to significantly increase the performance of application loops, if they can be vectorized. As with FLIX, using vector operations increases gate count, because the vector operations require additional execution logic to perform operations on multiple data elements in parallel. Also, the vector operations require a vector register file capable of holding data-element vectors.

The operation-fusion technique creates operations composed of several simpler operations. Using the fused operation in place of simpler operations reduces code size and instruction-issue bandwidth, and may also reduce register-file port requirements. Also, the latency of the fused operation may be less than the combined latency of the simple constituent operations.

An example of a fused operation is the add-with-shift-by-1 operation present in Tensilica's Xtensa LX base architecture. The add-with-shift-by-1 instruction shifts a value left by one bit and then adds it to another value. This instruction fuses a left-shift-by-a-constant operation and an add operation. One fused add-with-shift-by-1 operation replaces the two simpler operations, but it executes in one cycle.

Using fused operations may increase gate count if the fused operation requires additional logic, or if the fused operation requires additional register-file ports to access operands. However, fused operations that have constant inputs often have low incremental gate counts (the shift-by-1 in the above example is effectively free) while potentially providing significant performance benefits.

To realize the maximum performance improvement for a given gate count, XPRES considers custom processors that contain all combinations of vector operations, fused operations, and operations that combine those techniques (for example, a single operation that performs four parallel multiply-by-three-accumulate computations on two vectors of four integers, producing a result vector of four integers). In addition, XPRES uses FLIX to allow multiple independent operations to be issued and executed in parallel.

22. Low-Power SOC Design Using Xtensa LX

The Xtensa LX processor has been designed with numerous features to minimize power consumption. The principal mechanism for saving power is the use of clock gating, which disables register clocks when the individual registers are not active. This scheme saves switching power in the clock tree, in the registers, and in any logic driven by the registers.

The Xtensa LX processor uses another technique to save memory power by often disabling memories when they are not needed, as long as performance is not sacrificed. Both instruction and data memories are disabled whenever possible.

Finally, the designer can minimize power by making judicious configuration choices. For instance, the designer may choose smaller and fewer memories, the designer may avoid configuration options that require large amounts of logic that are not needed for an application, and the designer may configure extra instructions or design TIE instructions that more effectively execute an application so that the processor can spend more time in sleep mode. The best choices for power vary with the application.

22.1 Saving Power Through Clock Gating

The Xtensa LX processor allows two levels of clock gating. The first level of clock gating is based on global conditions. For instance, the WAITI option allows the processor to enter a sleep mode that turns off the clocks to almost all of the registers in the design. WAITI requires interrupts to be configured, because an interrupt wakes the processor from sleep mode. If interrupts are not configured, the `RunStall` signal can still be used to save power by allowing external logic to stall the processor pipeline and turn off the clock to many of the processor's registers. `WAITI` and `RunStall` are described in more detail in the following sections.

Other global conditions that allow the first level of clock gating to save power include instruction-cache line fills, and various conditions in the load/store units.

The second level of clock gating is functional clock gating, which is much more local to specific registers in the design. In many cases specific register clocks can be gated based on local conditions in the processor. Even though the processor as a whole is active, portions of the design can be deactivated and reactivated as needed to save power using this method. Choosing functional clock gating creates many smaller branches in the clock tree that are activated by individual enable signals. The Xtensa LX processor employs extensive clock gating for hardware added through designer-defined TIE constructs, which makes such extensions extremely power efficient.

For maximum power savings, both levels of clock gating should be enabled. However, if the designer only desires one level of clock gating due to design-flow restrictions, then just one (or neither) of the clock gating levels can be chosen. In general, if the processor will have long periods of inactivity, then the `WAITI` or `RunStall` options will save the most power. If the processor is mostly active, then the functional clock gating will save the most power. The designer must make choices based on the target application(s).

22.2 Wait Mode

The Xtensa LX processor's interrupt option adds the `PWaitMode` port and `WAITI` instruction to the Xtensa LX processor. When the `WAITI` instruction completes, it causes the processor to set the interrupt level in the `PS.INTLEVEL` register to the value encoded in the `WAITI` instruction. The processor then waits for all of the processor memory interfaces to become idle. After all of its memory interfaces become idle, the processor asserts `PWaitMode` and suspends operations until a non-masked interrupt occurs.

If the global clock gating option is enabled, then the majority of the clocks in the processor will be switched off while `PWaitMode` is asserted. If global clock gating is not enabled, then processor operations will still be suspended while `PWaitMode` is asserted, but there will be no power savings inside the Xtensa LX processor.

Inbound-PIF requests are not considered processor operations and can continue while the processor is in wait mode. The clock network to the inbound-PIF logic is gated by a separate set of conditions to allow the inbound-PIF logic to continue functioning while the processor is in wait mode.

22.3 Global Run/Stall Signal

The `RunStall` signal can be used to save power when there are no interrupts configured. `RunStall` shuts off most—but not all—of the processor's clock network, assuming first-level clock gating is configured. In particular, the clock to the PIF logic will not be gated off, nor will the clocks to some internal logic needed to complete any cache-line fills that are in progress. In general, local-memory enables will be deactivated, except when cache-line fills are completing or when inbound-PIF requests to instruction or data RAMs or the XLMI port are active. Because the `RunStall` signal switches off fewer clocks than in the WAITI interrupt-based power-saving scheme, power savings in general will be less. However, this technique does not require the use of interrupts.

`RunStall` can be activated at any time, but activating this signal stalls the internal processor pipeline. Thus, the designer is responsible for determining when it is appropriate to activate the external `RunStall` signal.

Index

Index