

8 Customizable Processors and Processor Customization

Steven Leibson

Tensilica, Inc.

Introduction

The first commercial microprocessor chip, Intel's 4004, appeared in November, 1971. Since then, most designers have used fixed-ISA (instruction-set architecture) processors in their system designs – first as processor chips in board-level designs and later as processor cores in SOCs. In fact, many system designers cannot envision designing a custom processor for their projects because the use of fixed-ISA machines has become so thoroughly engrained in the conventional design methodology. A small cadre of designers created custom processors based on bit-slice technology in the 1980s for applications with very high performance requirements, but electronic system design has largely evolved into an exercise in adapting standardized processor and DSP architectures to target tasks, often with additional hardware acceleration to bridge the inevitable gap between a task's required computations and the fixed-ISA processor's abilities.

Designers working for conventional processor vendors have always had the ability to extend their own processor ISAs as needed. For example, two features that distinguish DSPs from general-purpose processors are hardware MACs (multiplier/accumulators) and dual load/store units for XY-memory addressing. No intrinsic hardware limitation prevents processor designers from incorporating such features in their general-purpose processor ISAs. However, adding such resources increases the processor's silicon area (and therefore its cost) which may not be used by many of the applications of that processor. Successful DSPs must have these features. Successful general-purpose processors don't. Similarly, many other task-specific features are not good candidates for universal inclusion in general-purpose processor designs.

However, vendors of customizable processor IP (intellectual property) now offer SOC designers the tools needed to easily customize a processor's ISA for specific applications (and to automatically generate the associated software-development tools needed to program these processors), which produces processor cores that can execute specific, targeted tasks in many fewer clock cycles. This execution efficiency can be used to either increase a processor's execution "reach" without boosting its clock frequency or it can be used to reduce the maximum required clock rate for a given set of tasks, thus lowering the power and energy dissipation required to execute the target tasks.

The flexible silicon resources in high-gate-count SOC's now permit the use of customized processors in a wide variety of products, ranging from low-cost consumer products including MP3 players, video games, and printers to high-end products such as metropolitan-area network routers. Customized processors incorporate task-specific registers and task-specific hardware acceleration in the processor's execution pipeline instead of employing accelerators external to the processor. This move brings the acceleration hardware into intimate contact with the processor's registers, register files, memory-management hardware, and memory interfaces, which greatly improves the efficiency and throughput of data movement into and out of the accelerators and the added task-specific hardware greatly improves the processor's efficiency (measured in clock count) on the task's computations.

For example, MP3 players and other products that play and record audio may benefit from the use of a processor with audio-specific extensions such as one or two audio MACs. Video products such as camcorders and DVD players can benefit from processors that can directly manipulate RGB data as a native data type and networking products including network interfaces, switches, and routers benefit from processors with packet-processing hardware extensions. These abilities are all made possible by the flexibility available in the use of the SOC design methodology and the plastic nature of ASIC silicon.

This chapter is divided into two parts. The first part is a general discussion of the abilities and benefits of using customizable processors. The second part discusses the specifics of one commercial customizable processor: Tensilica's Xtensa family.

A benefits analysis of processor customization

In a generic 90 nm standard-cell foundry process, silicon density routinely exceeds 200K usable gates per mm². Consequently, a low-cost chip

(measuring 50 mm²) can carry 10M gates of logic. Simply because it's possible, some designer working on an SOC development team somewhere will find ways to exploit this potential in any given market. Designers face a set of daunting challenges however, because the traditional HDL-based approach to designing SOC hardware is failing:

- **Design effort:** In the past, silicon capacity and design-automation tools limited the practical size of a block of RTL to smaller than 100K gates. Blocks of 500K gates are now within the capacity of today's tools, but existing design methods are not keeping pace with silicon fabrication capacity, which now routinely puts many millions of gates on an SOC.
- **Verification difficulty:** The internal complexity of a typical logic block – hence the number of potential design bugs – grows far more rapidly than does its gate count. Consequently, verification complexity has grown disproportionately. Many teams that have developed real-world SOC designs report that they now spend as much as 90% of their development effort on verification.
- **Cost of fixing bugs:** The cost of fixing an SOC design bug is rising. Higher staff costs caused by growing design teams, 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 gets blamed for overall program delays.
- **Complexity and change 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 the creation of complex new standards such as the IPv6 Internet Protocol packet forwarding, G.729 voice coding, JPEG2000 image compression, MPEG4 video, and Rijndael AES encryption. These new protocol standards have much greater computational demands than their predecessors.

Although general-purpose, firmware-programmable embedded processor cores with fixed ISAs can handle many tasks, they often lack the bandwidth needed to perform complex data-processing tasks such as network packet processing, video processing, and encryption. To meet aggressive performance goals, chip designers have long turned to hardwired logic to implement these key functions. As the complexity and bandwidth

requirements of electronic systems increase, the total amount of logic rises steadily.

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

- How do SOC design teams ensure that the chip specification really satisfies customer needs?
- How do SOC design teams ensure that the chip really meets those specifications?

Further, a good SOC design team will also anticipate future needs of current customers and potential future customers – it has a built-in road map for the SOC design.

If the design team fails to create an appropriate SOC specification, the chip may work perfectly but it will not sell well enough to justify the design and manufacturing costs. Changes in the chip's requirements may be driven by demands of specific key customers, or may reflect rapid changes that frequently occur in the market such as the emergence of new data format standards or new feature expectations across an entire product category. While most SOC designs include some form of embedded control processor (usually with a fixed ISA), the limited performance of these general-purpose processors often precludes them from being used to meet the performance requirements of essential on-chip data-processing tasks so task-specific hardware is designed, which means that firmware often cannot be used to add or change fundamental new features. If such an SOC's functional abilities need to change, the chip must be redesigned. That is a costly path.

If the SOC design team fails to meet the specifications of the chip's design, additional time and resources must go towards changing or fixing the design errors. This resource diversion delays market entry and causes companies to miss key customer commitments. This sort of failure usually surfaces 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 relatively limited verification coverage of gate-level simulation. The underlying cause of the error might be a subtle bug in one design element or it might be a miscommunication of requirements – caused by subtle differences in assumptions between hardware and firmware teams, between design and verification teams, or between the SOC designer and SOC library or foundry supplier. In any case, the design team is often forced into an urgent cycle of chip re-design, re-verification and re-fabrication. SOC design “spins” rarely take less than six months, causing significant disruption to product and business plans.

Using microprocessor cores in SOC design

There are two closely related problems for system developers. One is to develop system designs with significantly fewer resources by making it much, much easier to design the chips in those systems. The second problem is making SOC's sufficiently flexible so 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, or 100, or 1000 different system designs while giving up none or, at most, a few of the benefits of integration. Solving these problems produces off-the-shelf SOC's that satisfy the requirements of next-generation system designs. This design approach amortizes the costs of chip development over a large number of system designs.

The specialized nature of individual embedded applications creates two issues for general-purpose embedded processor cores executing data-intensive tasks. First, there is a poor match between the critical functions of many embedded applications (e.g. image, audio, and protocol processing) and a processor's basic integer ISA (instruction set and register file). As a result of this mismatch, critical embedded applications require more computation cycles when implemented in firmware running on general-purpose embedded processor cores. Second, specialized embedded devices cannot take full advantage of a general-purpose processor's broad capabilities. Consequently, expensive silicon resources built into the processor core are wasted because they're not needed by the specific embedded task that's assigned to the processor.

Most embedded systems interact closely with the real world or communicate complex data at high rates. The associated data-intensive tasks in these applications could be performed by some hypothetical general-purpose microprocessor running at tremendous speed. This is indeed the approach employed in the personal computer market, with the result that the processors used in PCs cost hundreds of dollars and dissipate tens of watts. For embedded applications, expensive and power-hungry processors are not an appropriate alternative.

Instead, designers traditionally turn to hard-wired circuits to perform these data-intensive functions. In the past 10 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.

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 while logic blocks based on configured processor cores realize their state machines with firmware, which makes these blocks far more flexible than hard-wired RTL designs.

Benefiting from microprocessor extensibility

A fully featured configurable and extensible processor consists of a processor design and a design-tool environment that allows a designer to adapt the base processor design by altering or adding major processor functions, thus tuning the processor core to specific application requirements. 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 SOC design team to include features never considered or imagined by processor’s original designers.

Changing the processor’s instruction set, memories and interfaces can significantly improve the core’s efficiency and performance, particularly for the data-intensive applications that represent the “heavy lifting” for many embedded systems. These task-specific features might be too specific to justify inclusion in a general-purpose processor’s ISA. General-purpose processors are the result of many design compromises where features that provide modest benefits to all customers supercede features that provide dramatic benefits to a few. All design compromise is necessary because the historic costs and difficulty of processor design mandate that only a few different such cores can be built. However, automatic generation of a configurable processor and its associated software tool suite reduces the cost and development time so that inclusion of application-specific features and deletion of unused features suddenly becomes attractive for SOC design [162].

A *configurable processor* is a processor whose features can be pruned or augmented by parametric selection. Configurable processors can be implemented in many different hardware forms, ranging from ASICs with

hardware implementation times of many weeks, to FPGAs with implementation times of just minutes. *Extensible processors* – processors whose functions, especially the instruction set, can be extended by the application developer to include features never considered by the original processor designer – are an important superset of configurable processors.

For both configurable and extensible processors, the usefulness of the configurability and extensibility is strongly tied to the automatic availability of both hardware implementation and the software environment. Automated generation of software tools that support the core's extended features is especially important. Configuration or extension of the processor's hardware without synchronized enhancement of the compiler, assembler, simulator, debugger, real-time operating systems, and other software support tools would leave the promises of performance and flexibility through configurability unfulfilled, because the new enhanced processor could not be programmed very easily.

Table 8.1. Processor configuration and extension types.

ISA	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 with varying word widths –High-performance arithmetic and DSP (e.g. compound DSP instructions, vector/SIMD instructions, floating-point instructions), often with wide execution units and registers –Multiple independent operations per instruction –Selection among function unit implementations (e.g. small iterative multiplier vs. pipelined array multiplier)	–Memory-bus width –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 –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

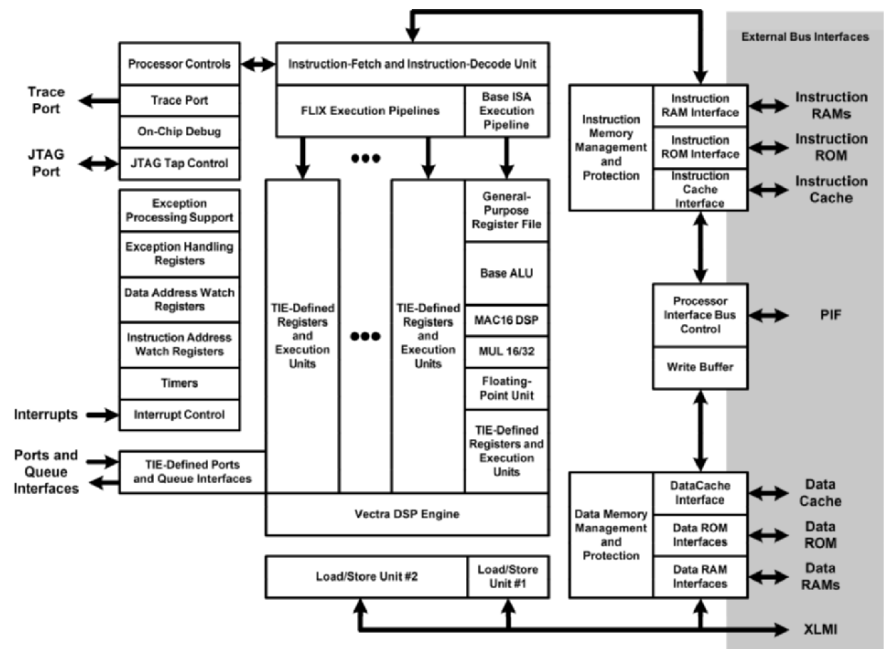


Fig. 8.1. Block diagram of configurable Xtensa processor.

Extensibility’s goal is to allow ISA features to be added or adapted in any form that optimizes the cost, power, and application-performance of the processor core. In practice, the configurable and extensible features can be broken into four categories, with examples, as shown in Table 8.1.

A block diagram for Tensilica’s configurable, extensible Xtensa processor appears in Figure 8.1. The figure identifies baseline ISA features, scaleable register files, memories and interfaces, optional and configurable processor peripherals, selectable DSP coprocessors, and facilities to integrate user-defined instruction-set extensions.

Processor extensibility serves as a particularly potent form of configurability because it handles a wide range of applications and is easily used by designers with a wide range of skills. Processor extensibility allows a system designer or application expert to directly exploit proprietary insight about the application’s functional and performance needs directly in the processor core’s instruction-set and register extensions.

How microprocessor use differs between SOC and board-level design

Hardwired RTL design has many attractive characteristics – small area, low power, and high-throughput. However, the liabilities of RTL (difficult design, slow verification, and poor scalability to complex problems) are starting to outweigh the benefits in large SOC designs. A design methodology that retains most of the efficiency benefits of RTL while reducing design time and risk has a natural appeal. Application-specific processors as a replacement for complex RTL fit this need.

An application-specific processor can implement data-path operations that closely match those of RTL functions. Equivalents of the RTL datapaths are implemented using the integer pipeline of the base processor, plus additional execution units, new registers and register files, and other functions added by the chip architect for a specific application. For Tensilica's Xtensa processor cores, those extensions are defined in a Verilog-like language called TIE, which is optimized for the 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 new processor instructions and registers described in TIE are available to the firmware programmer via the same compiler and assembler that target the processor's base ISA. All sequencing of operations within the processor's datapaths is controlled by firmware, through the processor's existing instruction fetch, decode and execution mechanisms. The firmware is written in a high-level language such as C or C++.

Extended processors used as RTL-block replacements routinely use the same structures as traditional data-path-based RTL blocks: deep pipelines, parallel execution units, problem-specific state registers, and wide data paths to local and global memories. These extended processors can sustain the same high computation throughput and support the same low-level data interfaces as typical RTL designs.

Control of the extended-processor datapaths is very different however. Cycle-by-cycle control of the processor's datapaths is not fixed in hardwired state transitions. Instead, the sequence of operations is explicit in the

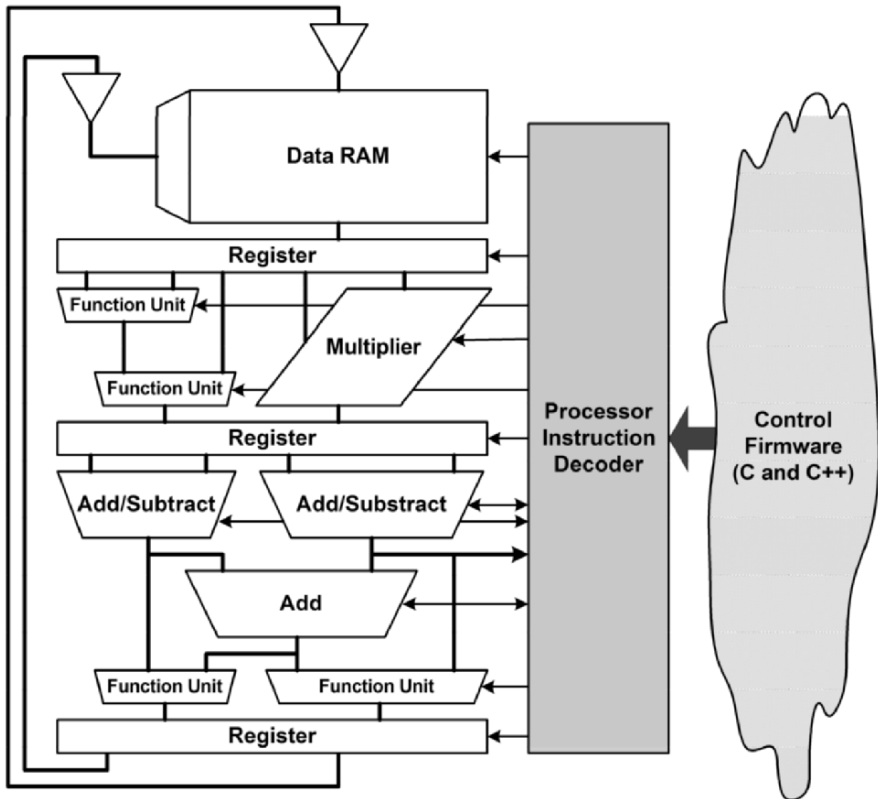


Fig. 8.2. Programmable function: data-path + processor + software.

firmware executed by the processor (shown in Figure 8.2). 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.

This design migration from hardwired state machine to firmware program control has important implications:

- **Flexibility:** Chip developers, system builders, and end-users (when appropriate) can change the block's function just by changing the firmware. The need for silicon respins is greatly reduced.
- **Firmware-based development:** Developers can use sophisticated, low-cost software-development methods for implementing 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 for extended processors run at hundreds of thousands or millions of cycles per second on instruction-set simulators.

- **Unification of control and data:** No modern system consists solely of hardwired logic. There is always a processor and some software. Moving functions previously handled by RTL into a processor removes the artificial distinction between control and data processing.
- **Time-to-market:** Moving critical functions from RTL to application-specific processors simplifies the SOC design, accelerates system modeling, and pulls in finalization of hardware. Firmware-based state machines easily accommodate changes to standards, because implementation details are not “cast in stone.”
- **Designer productivity:** Most importantly, migration from RTL-based design to the use of application-specific processor cores boosts the engineering team’s productivity by reducing the engineering resources needed to manually code and verify RTL hardware. A processor-based SOC design approach sharply cuts risks of fatal logic bugs and permits graceful recovery when (usually not if) a design bug appears or a specification change occurs.

Three examples, ranging from simple to complex, serve to 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, is the GSM audio codec used in cell phones. Profiling the codec code using an unaugmented RISC processor revealed that out of more than 200 million processor cycles, 80% of the cycles were devoted to executing multiplications. The simple addition of a hardware multiplier therefore produces a substantial acceleration of this codec implementation. Tensilica’s Xtensa processor offers a multiplier as a configuration option. That means that a designer can add a hardware multiplier to the processor’s data path and multiplication instructions to the processor’s instruction set simply by checking a box in a configuration page of the Xtensa processor generator.

The addition of a hardware multiplier cuts the number of cycles needed to execute the GSM audio codec code from 204 million to 28 million cycles, a 7x improvement in execution time. Adding a few more gates to the processor pipeline by selecting a MAC instead of a multiplier further reduces the number of cycles needed to execute the audio codec code to 17.9 million. These 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

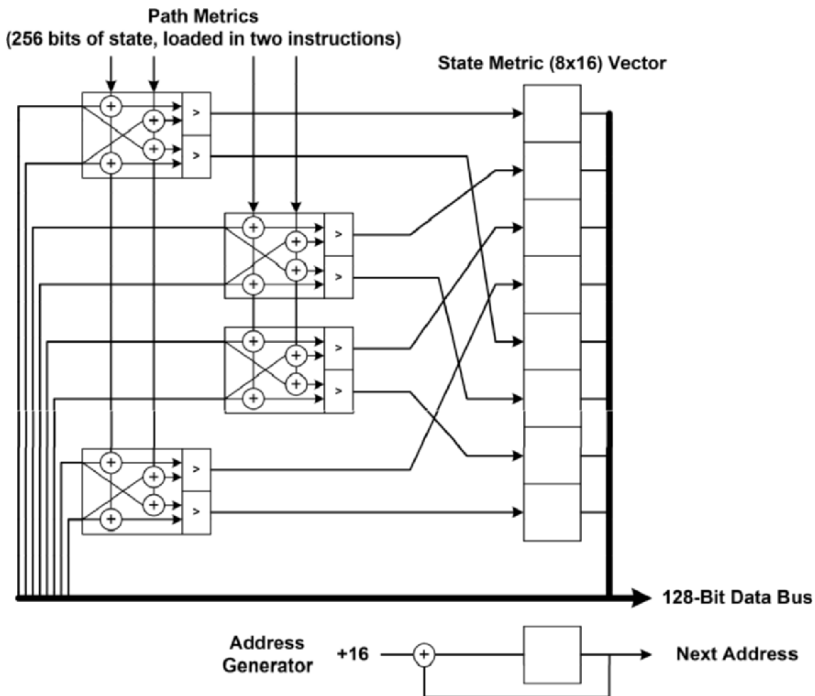


Fig. 8.3. Viterbi butterfly pipeline hardware.

out of a noisy transmission channel. This decoding scheme employs “butterfly” operations consisting of eight logical operations (four additions, two comparisons, and two selections) and uses eight 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. The TIE language allows a designer to add a Viterbi butterfly instruction to the Xtensa processor’s ISA, which uses the processor’s configurable 128-bit I/O bus to load data for eight symbols at a time, adds the pipeline hardware shown in Figure 8.3, and results in an average butterfly execution time of 0.16 cycles per butterfly. An unaugmented Xtensa processor executes Viterbi butterflies in 42 cycles, so the butterfly hardware adds only 11,000 gates and yet it achieves a 250x speed improvement over the out-of-the-box Xtensa processor.

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 used for motion estimation employs a SAD

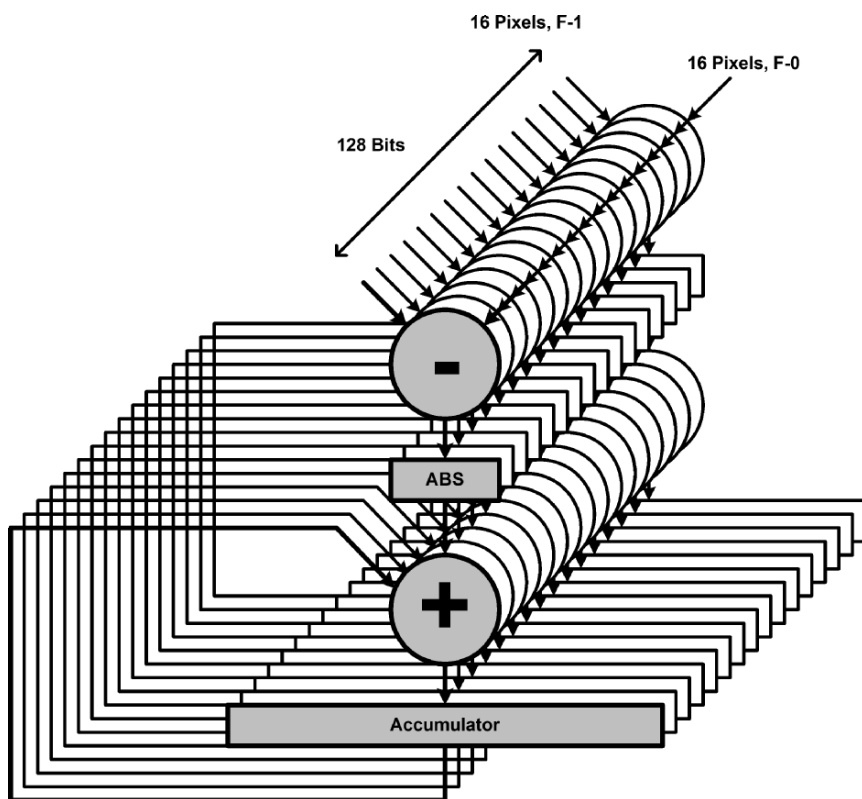


Fig. 8.4. A SIMD SAD computational extension reduces the computational load on the processor core by 46x.

(sum of absolute differences) operation consisting 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) video frame and a 15 frames/second image rate, the SAD operation requires slightly more than 641 million operations/second. As shown in Figure 8.4, it is possible to add SIMD (single-instruction, multiple-data) SAD hardware capable of executing 16 pixel-wide SAD instructions per cycle using TIE. (Note: Using the Xtensa processor's 128-bit maximum bus width, it is also possible to load all 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 number of cycles required to execute the algorithm from 641 million to 14 million operations/second – a substantial reduction.

Tensilica's extensible Xtensa processor core

ISA extension should be used to make a good ISA better, not to fix an ISA that's poorly suited to SOC design in the first place. Tensilica's Xtensa architecture was designed from the start to serve as an on-chip microprocessor core. Consequently, it is a small, fast processor core. The Xtensa ISA follows the RISC heritage traced back to IBM's 801 project in the 1970s and the RISC microprocessor work of John Hennessy at Stanford University and David Patterson at the University of California at Berkeley in the 1980s. The result of this early research produced small, fast processor architectures with several characteristic features:

- Load/store architecture
 - no memory references except for load and store instructions
- 3-operand instruction orientation
 - two operand sources, one result destination
- Large general-purpose register file
 - supports the load/store architecture
- Single-cycle instructions
 - for simplicity and speed
- Pipelined operation
 - produces single-cycle instruction throughput

The Xtensa ISA shares all of these characteristics with other RISC processor architectures but the architects of the Xtensa ISA realized that memory footprint would be critically important for on-chip processors using on-chip SOC memory (on-chip SOC memory is much more expensive than memory contained in standard memory chips), so the Xtensa architecture deviates from traditional RISC fixed-size instructions to reduce the firmware's memory footprint: the basic Xtensa ISA contains a mix of 16- and 24-bit instructions. These 16- and 24-bit instructions all perform 32-bit operations, so they are just as powerful as the 32-bit instructions of the older RISC architectures – they're merely smaller, which reduces program size and therefore reduces on-chip memory costs.

The original RISC processors employed fixed-size instructions of 32 bits to simplify the processor's fetch/decode/execute circuitry but the mechanism that converts incoming instruction words into 16- and 24-bit instructions in the Xtensa processor does not require complex logic. The amount of memory saved through the use of 16- and 24-bit instructions more than compensates for the gates used to implement the processor's mixed-size instruction-fetch and instruction-decode unit.

Because an Xtensa processor's instruction-fetch cycle retrieves more than one instruction per cycle (including fractions of an instruction word) and because a single Xtensa instruction can cross aligned fetch (word) boundaries, the Xtensa processor stores fetched words from the instruction stream in a FIFO holding/alignment buffer. For the base Xtensa ISA, the instruction buffer is 32-bits wide and two entries deep. It can be deeper and wider for certain Xtensa configurations. In addition to supporting the 16- and 24-bit instructions in the Xtensa processor's base ISA, this mixed-size instruction-fetch and instruction-decode unit also supports extended 32- and 64-bit, multi-operation instructions that can be added to the configurable Xtensa processor.

Xtensa configurable registers and register files

The Xtensa processor's base ISA incorporates the following register files and registers:

- A 32-bit, general-purpose register file that employs register windows
- A 32-bit program counter
- Various special registers

Figure 8.5 shows the processor's general-purpose 32-bit register file. This file, called the AR register file, has either 32 or 64 entries (a configurable attribute). Xtensa instructions access this physical register file through a sliding 16-register window. Register windowing allows the processor to have a relatively large number of physical registers while restricting the number of bits needed to encode a source or destination operand address to 4 bits. Thus a 3-operand Xtensa processor instruction needs only 12 bits to specify the registers holding the instruction's three operands.

Register windowing is a key ISA feature that allowed the Xtensa architects to achieve the small instruction sizes needed to minimize the application firmware's footprint while allowing for a large general-purpose register file, which boosts compiler efficiency and improves the performance of compiled code. Xtensa function calls and call returns slide the register window up and down the physical register file. Because window movement is restricted to a maximum of 12 register entries per call, some register entries in the physical general-purpose register file are shared between the current register window and the previous window. This overlap provides a set of register entries that can be used to automatically pass argument values and return function values between a calling function and the called function. XCC – the Xtensa C/C++ compiler – automatically uses the features of this register-windowing scheme to minimize the number of instructions used in function calls.

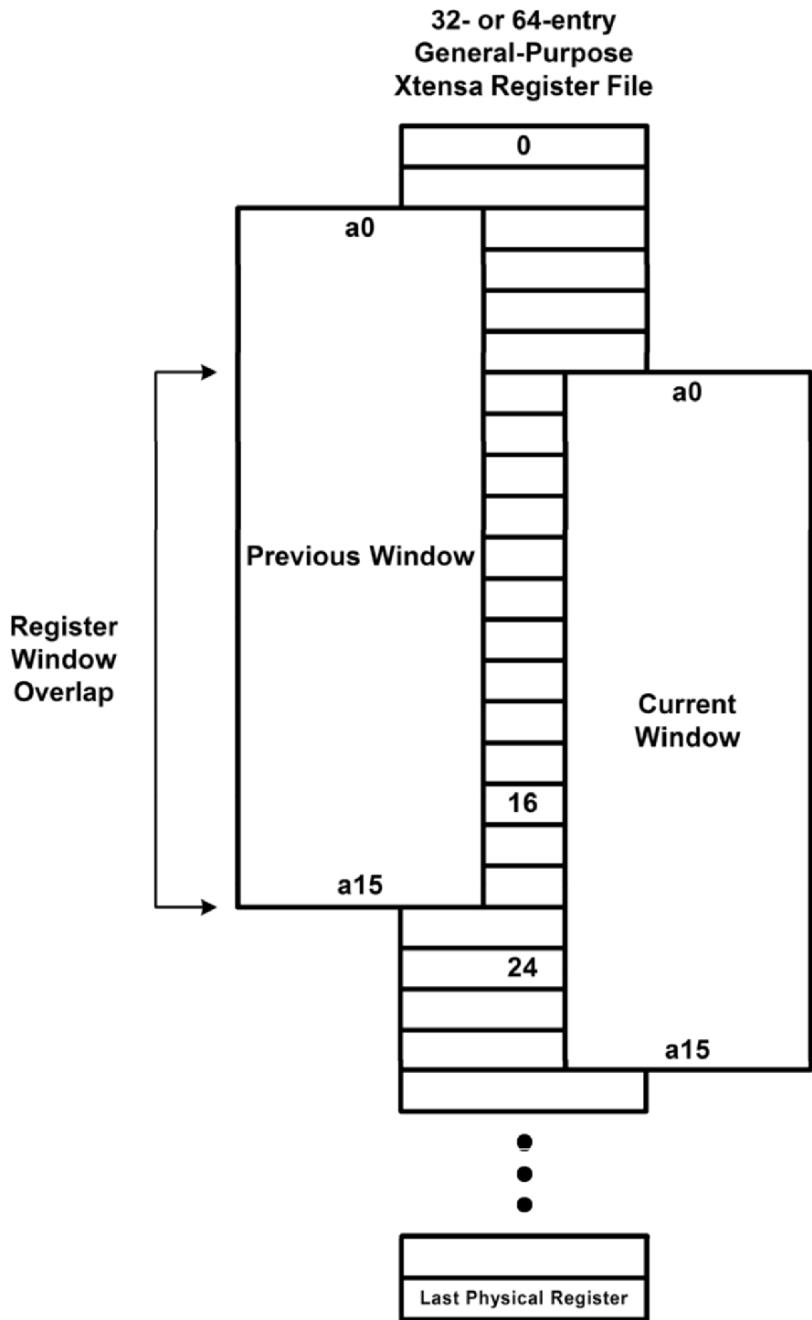


Fig. 8.5. The Xtensa ISA incorporates a 32- or 64-entry register file with register windows. Each entry in the file is a 32-bit, general-purpose register.

The Xtensa program counter

The Xtensa ISA employs a 32-bit program counter, which encompasses a 4-Gbyte address space. During a function call, only the lower 30 bits of the return address are saved, which restricts nested subroutines using function calls to a 1-Gbyte address space. (The other two address bits are used to store the amount of register-window translation: 0, 4, 8, or 12 entries.) A function-call return restores the saved 30-bit return address and leaves the upper 2 bits of the program counter untouched. Thus related sets of function calls can operate in any of the four 1-Gbyte address spaces that comprise the overall 4-Gbyte space. Jump instructions, which use 32-bit target addresses stored in register-file entries, shift program execution across the 1-Gbyte boundaries. This scheme reduces the number of memory-accesses required to implement function calls, which improves the overall performance of the Xtensa ISA.

The resulting 1-Gbyte address restriction that the Xtensa ISA imposes on linked subroutines puts no real practical limits the design of current-generation SOC, which almost universally employ per-processor code footprints much smaller than 1 Gbyte. Per-processor SOC code footprints are unlikely to approach 1 Gbyte for many, many years. That's simply too much code to run on any one embedded processor core in an SOC.

Memory address space

Unlike practical code-size limits, data-space requirements for SOC, seem to grow without limit, especially for media-oriented chips. Consequently, the Xtensa ISA provides full, unrestricted 32-bit data-space addressing for its load and store instructions. The Xtensa ISA employs a Harvard architecture that physically separates instruction and data spaces, although they share the same 4-Gbyte address space.

An Xtensa processor's local memories are divided into instruction and data memories and the processor employs separate instruction and data caches. The existence of local memories and caches, the address spaces allocated to the local memories, and the width of the bus interfaces to these local memories are all configuration options for Xtensa processors. Load, store, and fetch operations to addresses that are not allocated to local memories (as defined in the processor's configuration) are directed to the Xtensa processor's main bus interface, called the PIF (processor interface). The existence and width of a processor's PIF bus is another Xtensa configuration option.



Fig. 8.6. Little-endian byte ordering for a 32-bit word.

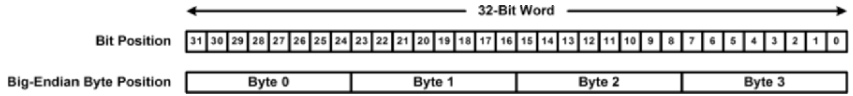


Fig. 8.7. Big-endian byte ordering for a 32-bit word.

Configurable byte ordering

As a configuration option, Xtensa processors support either big- or little-endian byte ordering, shown in Figures 8.6 and 8.7. Little-endian byte ordering stores a number’s low-order byte in the right-most byte of the 32-bit word and the high-order byte in the left-most byte. (The least-significant or “little” end comes first.) For example, a 4-byte integer is stored using the little-endian format in a 32-bit memory word as shown in Figure 8.6.

Big-endian byte ordering stores a 4-byte integer’s high-order byte in a memory location’s right-most byte and the low-order byte in the left-most byte. (The most-significant or “big” end comes first.) The long integer would then appear as shown in Figure 8.7.

Fast endian conversion with one instruction extension

There are a seemingly endless number of arguments regarding the relative merits of big- and little-endian byte ordering. Many of these arguments stem from religious discussions about the merits of the IBM PC vs. the (pre-Intel) Apple Macintosh computers because the underlying Intel and Motorola processors in those computers used opposite byte orderings.

In reality, both byte-ordering formats have advantages. Little-endian assembly language instructions pick up a multi-byte number in the same manner for all number formats. Because of the one-to-one relationship between address offset and byte number (offset 0 is byte 0), multiple precision math routines are correspondingly easy to write for little-endian processors. Numbers stored in big-endian form, with the high-order byte first, can easily be tested as positive or negative by looking at the byte value stored at offset zero, no matter the number of bytes contained in the number format. Software doesn’t need to know how long the number is nor does it need to skip over any bytes to find the byte containing the sign

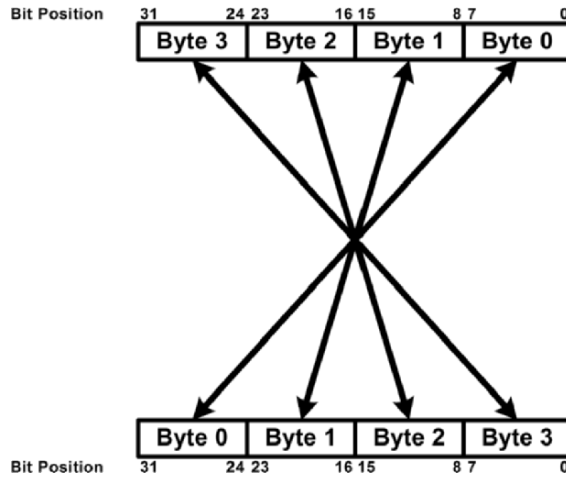


Fig. 8.8. Conversion from one endian format to the other for a 32-bit word.

information. Numbers stored in big-endian format are also stored in the order in which they are printed out, so binary-to-decimal conversion routines are particularly efficient.

The existence of “endian” issues means that whenever multi-byte values are written to a file, the software must know whether the underlying processor is a big- or little-endian machine and whether the target file format is big- or little-endian. For example, if a big-endian machine operates on a standard-format, little-endian file, the software must first reverse the byte order of all the multi-byte values in the file or the file format will not adhere to the file-format standard. SOC designers developing chips that must deal with standard file formats must be concerned with byte ordering because many of these formats have an established byte ordering. Any software that deals with these file formats must correctly deal with the format’s byte ordering. Figure 8.8 illustrates how bytes are swapped in a 32-bit word to convert from one endian format to the other.

Audio and video media streams come in both endian formats as well so nearly any SOC dealing with multimedia files will need to work with both byte-ordering formats. Consequently, the choice of a processor’s endian orientation is usually not critical because the processor will need to make byte-ordering conversions in any case. The following line of C code converts from one endian format to the other:

```
unsigned ss = (s<<24) | ((s<<8) & 0xff0000) | ((s>>8) & 0xff00) | s>>24;
```

The Xtensa C/C++ compiler (XCC) translates the line of C above into nine base Xtensa assembly-language instructions as follows (the input

word is passed in register a14 and the endian-converted result ends up in register a10):

slli a9, a14, 24	Form intermediate result bits 24-31 in a9
slli a8, a14, 8	Shift 32-bit word left by 8 bits, save in a8
srlr a10, a14, 8	Shift 32-bit word right by 8 bits, save in a10
and a10, a10, a11	Form intermediate result bits 9-15 in a10
and a8, a8, a13	Form intermediate result bits 16-23 in a8
or a8, a8, a9	Form intermediate result bits 16-31 in a8
extui a9, a14, 24, 8	Extract result bits 0-7, save in register a9
or a10, a10, a9	Form result bits 0-15, save in register a10
or a10, a10, a8	Form final result in register a10

The C-level conversion code could execute nine times faster if the base Xtensa ISA had one instruction that performed 32-bit endian conversion. However, a fundamental tenet of ISA development is to minimize a processor's gate count by keeping the base ISA as lean as possible and not larding it with instructions that might only be useful in some applications. Not all processors need to perform endian conversions. While such specialized instructions may improve the processor's performance for a few application programs, they increase the processor's size for every SOC design using that processor. ISA design for *configurable* processor cores avoids the addition of specialized instructions to the processor's base ISA by allowing each SOC design team to add task-specific instructions needed to meet project-specific design and performance goals.

Specialized instructions easily can be added to an Xtensa configurable processor by the SOC designer using TIE. For example, defining a BYTESWAP instruction that takes a 32-bit word from one of the processor's general-purpose register-file entries, converts the word from one endian format to the other, and stores the result in another register-file entry is a simple task. The TIE description to create this new instruction is remarkably short:

```
operation BYTESWAP {out AR outR, in AR inpR} { }
{
    wire [31:0] reg_swapped =
        {inpR[7:0],inpR[15:8],inpR[23:16],inpR[31:24]};
    assign outR = reg_swapped;
}
```

The interface to the BYTESWAP instruction is defined by the information contained in the first set of curly braces of the operation section. Within the first set of braces, the argument *outR* specifies the destination entry in the AR register file for the result of the instruction. Argument *inpR* specifies the AR register-file entry that provides the source operand

for the instruction. The second set of braces in the operation statement can be used to specify additional internal states for this *operation* extension, but this TIE feature is not used in this example.

The behavior of the BYTESWAP instruction is defined within the next set of curly braces. The first line in this group defines how the new machine instruction should compute the byte-swapped 32-bit value and assigns the result of the operation to a temporary variable named *reg_swapped*. Note that the values of the intermediate wires, states, and registers will be visible in the tailored Xtensa debugger for a processor incorporating this BYTESWAP instruction. This feature greatly facilitates the debugging of TIE instructions in a software environment. The second line above assigns the byte-swapped value to the output argument *outR*.

From this single instruction description, the TIE Compiler within the Xtensa processor generator builds the necessary execution-unit hardware, adds it to the processor's RTL description, and adds constructs in the software-development tool suite so that the new BYTESWAP instruction can be used as an intrinsic in a C or C++ program.

The BYTESWAP instruction is an example of a fused instruction. Nine *dependent* instructions (each instruction in the sequence depends on results from previous instructions) have been fused into one. In this example, the circuitry required to implement the function is extremely simple. The execution unit for this instruction needs little more hardware than a few additional wires to scramble byte lanes, yet this new instruction speeds endian conversion by a factor of 9x. This example demonstrates that a small addition to a processor's hardware can yield large performance gains. In addition, the new BYTESWAP instruction doesn't use the intermediate-result registers that are used in the 9-instruction byte-swap routine. Some gates are added in the processor's instruction decoder to decode the new instruction, but these few gates do not make the processor core noticeably larger than the base processor.

The TIE language

In general, most new instructions described in TIE implement more complex operations than BYTESWAP. In addition to the *wire* statement used in the above example, the TIE language includes several operators and built-in functions to describe new instructions. These operators and function modules appear in Tables 8.2 and 8.3, respectively.

Table 8.2. TIE operators.

Operator Type	Operator Symbol	Operation
Arithmetic	+	Add
	−	Subtract
	*	Multiply
Logical	!	Logical negation
	&&	Logical and
		Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
	==	Equal
	!=	Not equal
Bitwise	~	Bitwise negation
	&	Bitwise and
		Bitwise or
	^	Bitwise ex-or
	^~ or ~^	Bitwise ex-nor
Reduction	&	Reduction and
	~&	Reduction nand
		Reduction or
	~	Reduction nor
	^	Reduction ex-or
	^~ or ~^	Reduction ex-nor
Shift	<<	Left shift
	>>	Right shift
Concatenation	{ }	Concatenation
Replication	{ { } }	Replication
Conditional	?:	Conditional
Built-in modules	<module-name>(...)	See Table 4.2

Improving application performance using TIE

As demonstrated above, TIE extensions improve the execution speed of an application running on an Xtensa processor by enabling the creation of single instructions that perform the work of multiple general-purpose instructions. Several techniques can be used to combine multiple general-purpose operations into one instruction. Three common techniques available through TIE are:

- Fusion
- SIMD/vector transformation
- FLIX

Table 8.3. Built-in TIE function modules.

Format	Description	Result Definition
TIEadd(a, b, cin)	Add with carry-in	$a + b + \text{cin}$
TIEaddn(a0, a1, ... an-1)	N-number addition	$a0 + a1 + \dots + a_{n-1}$
TIEcmp(a, b, sign)	Signed and unsigned comparison	$\{a < b, a \leq b, a == b, a \geq b, a > b\}$
TIEcsa(a, b, c)	Carry-save adder	$\{a \& b \mid a \& c \mid b \& c, a \wedge b \wedge c\}$
TIEmac(a, b, c, sign, negate)	Multiply–accumulate	Negate ? $c - a * b : c + a * b$ where sign specifies how a and b are extended in the same way as for TIEmul
TIEmul(a, b, sign)	Signed and unsigned multiplication	$\{\{m\{a[n-1] \& \text{sign}\}\}, a\} * \{\{n\{b[m-1] \& \text{sign}\}\}, b\}$ where n is size of a and m is size of b
TIEmulpp(a, b, sign, negate)	Partial-product multiply	negate ? $- a*b : a*b$
TIEmux(s, d0, d1, ..., dn-1)	n-way multiplexer	$s==0 ? d0 : s==1 ? d1 : \dots : s==n-2 ? d_{n-2} : d_{n-1}$
TIEpsel(s0, d0, s1, d1, ..., sn-1, dn-1)	n-way priority selector	$s0 ? d0 : s1 ? d1 : \dots : s_{n-1} ? d_{n-1} : 0$
TIEsel(s0, d0, s1, d1, ..., sn-1, dn-1)	n-way 1-hot selector	$(\text{size}\{S0\} \& D0) \mid (\text{size}\{S1\} \& D1) \mid \dots (\text{size}\{S_{n-1}\} \& D_{n-1})$ where size is the maximum width of D0 ... Dn-1

To illustrate these three techniques, consider a simple example where an application spends most of its execution time computing the average of two arrays in a loop:

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

The above C code adds two short data items and shifts the sum right by 1 bit in each loop iteration. Two base Xtensa instructions are required for each computation, not counting the instructions required for loading and storing the data. These two operations can be fused into a single TIE instruction:

```
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];
}
```

This fused TIE instruction, named *AVERAGE*, takes two input values (*input0* and *input1*) from entries in the AR register file, computes the output value (*res*), and then saves the result in another AR register-file entry. The semantics of the instruction, an add followed by a shift, are described above. A C or C++ program uses the new *AVERAGE* instruction 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]);
```

Assembly code can also directly use the *AVERAGE* instruction. The entire software tool chain recognizes *AVERAGE* as a valid instruction for processors built using this TIE extension.

Instruction fusion is not the only way to form new instructions for Xtensa processors. Two other techniques are SIMD (single-instruction, multiple-data) and FLIX (flexible-length instruction extensions), the Xtensa version of VLIW (very-long instruction word). SIMD instructions gang multiple, parallel execution units that perform the same operation on multiple operands simultaneously. This sort of instruction is particularly useful for stream-processing applications such as digital audio and video.

In the fused-instruction example shown above, one TIE instruction combines an add and a shift operation, cutting the number of instruction cycles for the overall operation in half. Other types of instruction combinations can also improve performance.

Adding SIMD instructions using TIE

The C program in the above example performs the same computation on a new data instance during each loop iteration. SIMD (single-instruction, multiple-data) instructions (also called vector instructions) perform multiple loop iterations simultaneously by performing parallel computations on different data sets during the execution of one instruction. TIE instructions can combine fusion and SIMD techniques. Consider, for example, a case where a TIE instruction computes four *AVERAGE* operations in one instruction:

```
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]};
}
```

Computing four 16-bit averages simultaneously requires that each data vector be 64 bits wide (containing four 16-bit scalar quantities). However, the general-purpose AR register file in the Xtensa processor is only 32 bits wide. Therefore, the first line in the SIMD TIE example above creates a new register file, called *VEC*, with eight 64-bit register-file entries that hold 64-bit data vectors for the new SIMD instruction. This new instruction, *VAVERAGE*, takes two 64-bit operands (each containing four 16-bit scalar quantities) from the *VEC* register file, computes four simultaneous averages, and saves the 64-bit vector result in a *VEC* register-file entry. 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 C/C++ compiler generated for a processor built with this TIE description automatically recognizes a new 64-bit C/C++ data type called *VEC*, which corresponds to the 64-bit entries in the new register file. In addition to the *VAVERAGE* instruction, the Xtensa processor generator automatically creates new load and store instructions to move 64-bit vectors between the *VEC* register file and memory. The XCC compiler uses these instructions to load and store the 64-bit vectors of type *VEC*. Compared to the fused instruction *AVERAGE*, the SIMD vector-fused instruction *VAVERAGE* requires significantly more hardware (in the form of four 16-bit adders) because it performs four 16-bit additions in parallel. The four 1-bit shifts do not require any additional gates.

The performance improvement gained by combining vectorization and fusion is significantly larger than the performance improvement from fusion alone. The addition of SIMD instructions to an Xtensa processor nicely dovetails with Tensilica's XCC C/C++ compiler, which has the ability to unroll and vectorize the inner loops of application programs. The loop acceleration achieved through vectorization is usually on the order of the number of SIMD units within the enhanced instruction. Thus a 2-operation SIMD instruction approximately doubles loop performance and an 8-operation SIMD instruction speeds up loop execution by about 8x.

Adding FLIX instructions using TIE

Although multiple operations occur simultaneously in a SIMD instruction, they are *dependent* operations. VLIW instructions bundle multiple *independent* operations into one machine instruction. Xtensa FLIX instructions are multi-operation instructions that allow a processor to perform multiple,

simultaneous, *independent* operations by encoding the multiple operations into a wide instruction word. Each operation within a FLIX instruction is independent of the others. The XCC compiler for Xtensa processors bundles these independent operations into a FLIX-format instruction as needed to accelerate code. While TIE-defined fused and SIMD instructions are 24 bits wide, FLIX instructions are either 32 or 64 bits wide, to provide enough instruction-word bits to fully describe the multiple independent operations. Xtensa instructions of different sizes (base instructions, single-operation TIE instructions, and multi-operation FLIX instructions) can be freely intermixed.

Consider again the *AVERAGE* example from above. Using base Xtensa instructions, the inner loop contains the *ADD* and *SRAI* instructions to perform the actual computation. Two *L16I* load instructions and one *S16I* store instruction move the data as needed and three *ADDI* instructions update the address pointers used by the loads and stores. A 64-bit FLIX instruction format with one operation slot for the load and store instructions, one slot for the computation instructions, and one slot for address-update instructions can greatly accelerate this code as follows:

```
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 defines an instruction format with three operation slots. The last three lines of code list base ISA instructions that are to be available in each slot defined for this FLIX configuration. Note that all the instructions specified are existing core instructions in the processor's base ISA, so their definition need not be provided in the TIE code because the Xtensa processor generator already knows about base Xtensa instructions.

For this example, the C/C++ program need not be changed at all. The processor generation creates a C/C++ compiler that will compile the original source code while automatically exploiting the FLIX extensions. 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    }
```

A computation that requires eight cycles per iteration on a base Xtensa processor now requires just three cycles per iteration, which is nearly a 3x performance increase. It took only five lines of relatively simple TIE code

to specify a FLIX configuration format with three instruction slots. Instruction fusion, SIMD, and FLIX techniques can be combined to further reduce cycle count.

Conclusion

Configurable processors are not the right choice for every hardware block on an SOC. 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. A FIFO controller built with a RAM and some wrapper logic can be emulated via memory operations within a processor but a simple hardware FIFO is faster and simpler.
- **Very deep pipelines:** Some computation problems have so much regularity and require very little state-machine control. For these tasks, a single very 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 processor cores could be used to control such deep pipelines, but the benefits of instruction-by-instruction control would be of less help in these applications, if the algorithms are well known and not subject to change.

The migration of functions from software to hard-wired logic over time is a well-known phenomenon. During early design exploration of pre-release protocol standards, processor-based implementations are common even for simple standards that clearly allow efficient logic implementations. Some common protocol standards that have followed this path include popular video codecs such as MPEG2, 3G wireless protocols such as W-CDMA, and encryption and security algorithms such as SSL and triple-DES. The speed of this migration, however, has been limited by the large gap in performance and design ease between software-based and RTL-based development.

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