cādence®

The What, Why, and How of Customizable Processors

Meeting performance, cost, and power objectives while reducing ASIC design risk and increasing design flexibility

Customizable processors that perform intensive data processing are designed to provide programmability in the performance-intensive dataplane of the system-on-chip (SoC) design. Not only do they combine the capabilities of a DSP and a CPU, but they can be customized to maximize efficiency for your target application.

Contents

Introduction

While processors are often used for the control functions in system-on-chip (SoC) designs, designers turn to RTL blocks for many data-intensive functions that control processors can't handle. However, RTL blocks take a long time to design and even longer to verify, and they are not programmable to handle multiple standards or designs.

The most common embedded microprocessor architectures—such as the ARM®, MIPS, and PowerPC processors—were developed in the 1980s for stand-alone microprocessor chips. These general-purpose processor architectures, or CPUs, are good at executing a wide range of algorithms with a focus on control code, but SoC designers often need more performance in critical datapath portions of their designs than these microprocessor architectures can deliver.

To bridge this performance gap, the two most-used approaches are to run the general-purpose processor at a higher clock rate (thus extracting more performance from the same processor architecture), or to hand-design acceleration hardware that offloads some of the processing burden from the processor. Running a general-purpose processor core at a high clock rate incurs a power and area penalty, and designing acceleration hardware takes additional development time, not just for the design but for verification of the new acceleration hardware. In fact, verification can consume as much as 70% to 80% of the total design time.

This white paper discusses how Cadence® Tensilica® Xtensa® customizable processors can achieve high performance and lower energy consumption, save time, and provide design flexibility versus hand-coded RTL hardware or a general-purpose processor.

Getting More Performance the Old Way #1—Higher Clock Speed

When designing a new chip, some ASIC design teams wait until they know exactly how much processing "horse-power" they need, then they select the processor. This design approach locks up the chip's system architecture by basing it around the relatively narrow performance range occupied by all of the general-purpose processor architectures. For example, all 32-bit RISC architectures provide essentially the same performance at the same clock speed.

If the ASIC design team is wrong about the system's performance needs and additional processor performance is required, the only alternative available when using general-purpose processors that have fixed instruction set architectures (ISAs) is to use a processor from the same family that runs at a higher clock rate. This is a tried-and-true approach to system engineering dating back to the earliest days of microprocessor usage.

However, this decades-old approach has clearly reached the end of its useful life. High clock rates, now measured in GHz, result in excessive power and energy consumption. Chips overheat, fans are needed, and power supply costs rise as a result of this additional energy consumption. All of these consequences increase system costs.

In addition, a general-purpose processor can just as easily be employed by one of your competitors in a similar design, making your software and design ideas that much easier to copy.

What is the alternative? Custom processors, such as the Xtensa processor, can be optimized for a specific application. Optimization adds a variety of useful capabilities that keep clock rates and energy consumption low:

- Special registers sized to the natural data types of the tasks to be performed

- Specialized execution units that efficiently perform task-specific algorithms, often in one or two clock cycles

- Customized, system specific I/Os that can connect directly to neighboring blocks of dedicated hardware

Further, no other company can replicate your application-specific processor. You can hide portions of your design or give others full access to it using the development tools. These advantages help keep the design pirates away.

Getting More Performance the Old Way #2—RTL Acceleration

Because many applications simply cannot run fast enough on standard embedded microprocessors, even with an auxiliary DSP, engineering teams often use a hardware description language (HDL) such as Verilog or VHDL to create hand-coded acceleration hardware for the parts of their design with aggressive performance goals that are clearly out of any microprocessor's reach. However, custom RTL hardware takes a long time to design and even longer to verify. In addition, RTL hardware blocks can't be easily changed once they're designed because of the huge verification time. Yet changes are often needed to fix bugs or to accommodate new standards or product features.

Figure 1 shows the internal structure of a generic RTL accelerator block. The block's datapath appears on the left and its state machine appears on the right.
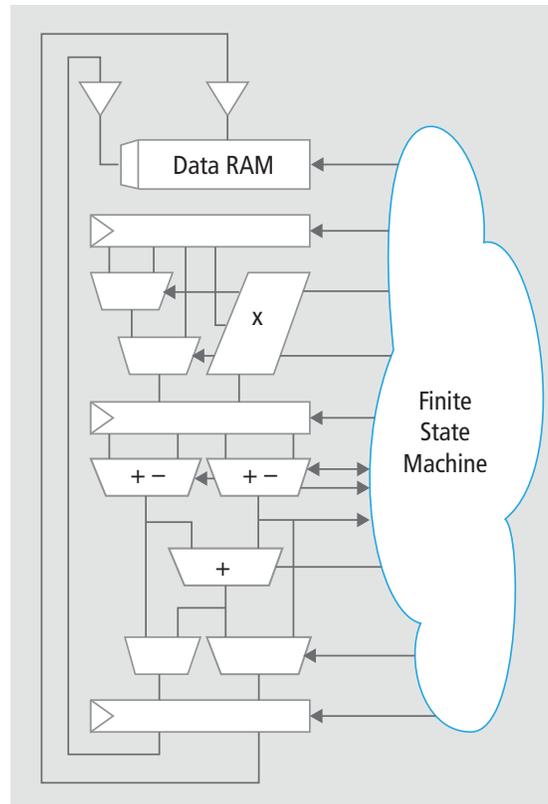
Figure 1: Hardwired RTL = Datapath Plus State Machine

In most RTL accelerators, the datapath consumes the vast majority of the gates. A typical datapath may be as narrow as 16 or 20 bits or range up to hundreds of bits wide, depending on the target task or application. RTL datapaths typically contain many data registers and often include significant blocks of RAM or interfaces to blocks of memory that are shared with other RTL blocks.

By contrast, the RTL logic block's finite state machine (FSM) contains nothing but the control logic. All the nuances of sequencing data through the datapath, all the exception and error conditions, and all the handshakes with other blocks are captured in the FSM. While the FSM is often small in area compared to the datapath, it most often embodies most or all of the design and verification risk due to its logical complexity. A late design change made to an RTL acceleration block is more likely to affect the FSM than the datapath because the FSM contains most of the design complexity.

An alternative to dedicated RTL blocks is using a custom processor for embedding the datapath logic and gates, and embedding the FSM control logic into the processor's firmware to decouple the datapath and the FSM control logic. This separation minimizes hardware design changes and gives flexibility to keep pace with updates that can easily be made in firmware. Custom processors support the RTL block's ability to have wide, non-integer-sized datapaths—which can be guaranteed correct-by-construction by the processor vendor—while reducing the risks associated with FSM design because processor-based FSMs are firmware programmable.

What is a Customizable Processor?

Xtensa customizable processors, based on a single processor architecture, use a common development flow with tools that scale from tiny microcontrollers and digital signal controllers to high-performance, real-time controllers and DSPs. Xtensa processors allow you to make the processor run your application more efficiently. You can tailor the Xtensa processor in various ways:

• Select from standard configuration options, such as bus widths, interfaces, memories, and preconfigured execution units (floating-point units, DSPs, etc.)

- Add new registers, register files, and custom task-specific instructions that support specialized data types and operations such as 48-bit data types (stereo audio sample pairs), 56-bit data and operations for security processing (encryption and decryption), or 256-bit data types and operations for packet processing

- Add interfaces for more input/output (I/O) bandwidth to move data into and out of the processor such as: Ports (general-purpose I/O [GPIO]), Queues (FIFO), and memory Lookup interfaces

- Add instructions on top of the base ISA from pre-defined options or create your own using the Tensilica Instruction Extension (TIE) language

Xtensa processors can implement wide, parallel, and complex datapath operations that closely match those used in custom RTL hardware. The equivalent datapaths are implemented by augmenting the base processor's integer pipeline with additional execution units, registers, and other functions developed by the chip architect for a target application

You get this customization with an automated flow that requires no extra processor verification and keeps the development tools and simulation models updated with every change. The processor is ultimately delivered as synthesizable RTL code, ready for integration into an FPGA prototype or SoC design, so they fit easily into existing design flows.

The result is a new processor, not a processor with bolted-on coprocessors. The new instructions and registers are available to the firmware programmer via the same compiler and assembler that target the processor's base instructions and register set. The instruction extensions greatly accelerate the processor's performance on the targeted algorithm and the controlling firmware can be written in a high-level language (C or C++) for easy development and maintenance.

Processor Design Cycle

The process for creating a processor's RTL description varies from vendor to vendor. Many vendors allow designers to manually insert hand-coded RTL instructions into the processor's RTL code, but this approach cannot provide mechanisms that guarantee the operational correctness of instructions that have been manually created and inserted. This lack of verification can be of great concern to designers that are new to processor design. In addition, the manual addition of instructions to a processor's ISA means that the associated software-development tools cannot know about, and therefore cannot exploit, the new instructions. As a result, the designer must have expertise in a higher-level programming language, such as C, as well as the assembly language to achieve a processor that yields high performance.

On the other hand, Xtensa processors offer another approach to customization: using a specialized language, Tensilica Instruction Extensions (TIE), to define the desired processor instructions and I/O. This language closely resembles a simplified version of Verilog, the hardware description language that ASIC designers already know. TIE enables you to extend the base RISC architecture for specific tasks, and permits the high-level specification of new datapath functions as new processor instructions, registers, register files, I/O ports, and FIFO queue interfaces.

A TIE description is both simpler and much more concise than RTL because it omits all sequential logic descriptions—including FSM descriptions and initialization sequences. These complex items are more easily developed in firmware. It is also unnecessary to describe the logic structure of these new functions and registers. The Xtensa Processor Generator (XPG) infers structure from the functional TIE description and creates new processor hardware that is guaranteed correct-by-construction.

The XPG creates an RTL description of the processor and generates tailored versions of all necessary software development tools including the compiler, assembler, debugger, and instruction set simulator, as well as a C or SystemC simulation model of the processor and EDA synthesis scripts. No manual work is required to match software development tools and processor. This new processor hardware is automatically blended into the Xtensa processor's base architecture, creating a seamless fusion of base architecture and task-specific ISA extensions.

The Xtensa development tools and flow automate processor creation, guaranteeing that the results are correct-by-construction. The development process includes the following steps:

1. Choose one of the Xtensa Reference Cores (XRCs) that most closely matches the expected requirements.

2. Compile representative C/C++ to check speed and power results on the chosen XRC using the Xtensa tools, including the Instruction Set Simulator (ISS) and Profiler, to identify areas that need improvement—all using the Eclipse-based Integrated Development Environment (IDE).

3. Try different configuration options to improve performance and re-profile.

4. Add new instructions and I/O using the TIE language and re-profile.

5. When the results meet your application needs, build the processor using our standard Xtensa Processor Generator (XPG) flow to get everything needed for your EDA development flow to tape out.

Achieving Lower Energy Consumption with the Xtensa Processor

How can a processor provide equivalent power consumption to optimized RTL? Don't processors, by definition, use more power?

For many tasks, the Xtensa processor can deliver performance comparable to an RTL accelerator block, running at equally low operating frequencies and therefore consuming similar power. So using an Xtensa processor to implement a task does not necessarily mean a large sacrifice in power for the benefit of programmability and faster development time.

When comparing processors, a focus on total energy consumption is the key. Too often, designers fixate on a static "milliWatts per megahertz" (mW/MHz) number while ignoring the total energy consumption of the workload. There is an implicit assumption in this type of thinking that all RISC processors deliver about the same performance per clock cycle. This assumption does not apply to processors that have been customized for a specific task or application.

For example, adding a few custom instructions, increases the processor core's size, which in turn increases the average power dissipation per clock cycle (increases the mW/MHz rating). However, if the custom instructions dramatically cut the total clock cycles required to perform a given workload (the target C code application), then the total energy consumed (power-per-cycle multiplied by total cycle time) can be substantially reduced.

Example: a 20% increase in power dissipated per clock cycle, offset by a 3X speed up in task execution, actually reduces energy consumption by 60%. This reduction in required task-execution cycles allows the system either to spend much more time in a low-power sleep state or to reduce the processor's clock frequency and core operating voltage, leading to further reductions in both dynamic and leakage power.

The base Xtensa Instruction Set Architecture (ISA), common to all Xtensa processors, provides the starting point for creating processors with the industry's lowest energy and highest performance when compared to legacy fixed-ISA processors. Xtensa processors are fully customizable, and you can add application-specific instructions using the automated processor generator. Consequently, it is important to compare equivalent processor configurations when comparing the Xtensa processor IP to competing general-purpose, fixed-ISA processor IP offerings.

Benefits of Using a Custom Processor Design

Customizing your processor design makes the processor unique, which makes it much harder for competitors to copy your ideas. You get a version of a processor that no one else can buy. No one else can get the matching software tool chain unless you provide it to them, so no one can use the processor's custom additions in your ASIC unless you allow it. When used with your software development tools, your optimized processor will get better performance, operate at lower required clock rates, and consume less energy than the industry-standard, fixed-ISA microprocessors.

See the *Ten Reasons to Customize a Processor IP* white paper for more information.

Customizable Processors for Digital Signal Processing

Many designs use a standard 32-bit processor coupled with a separate DSP to accelerate digital signal processing. However, using two processors means that data must transfer between the processor and DSP over some sort of interconnect, usually a standard bus, which slows performance. Xtensa processors do not need a separate DSP, as the DSP functions can be built right into the processor itself, eliminating inter-processor data transfers over a slow processor bus. For a more in-depth look at this aspect of SoC design, see "Example 1: Accelerating the Fast Fourier Transform (FFT) with the Xtensa Processor" in this white paper.

Customizable Processors as RTL Alternatives

You can use custom processors, such as Xtensa processors, as an alternative to hand-coded RTL blocks by adding the same datapath elements as implemented in RTL accelerator blocks. These datapath elements include deep pipelines, parallel execution units, task-specific state registers, and wide data buses to local and global memories. This allows the custom processors to sustain the same high computation throughput and support the same data interfaces as RTL hardware accelerators. You can optimize a customizable processor to run your application more efficiently.

Processor datapaths are controlled differently from their RTL counterparts. Control of a processor's datapaths is not frozen in a hardware FSM's state transitions. Instead, the processor-based FSM is implemented in firmware (see Figure 2), which greatly reduces the amount of effort needed to fix an algorithm bug or to add new features. In a firmware-controlled FSM, control-flow decisions occur in branches, load and store operations implement memory accesses, and computations become explicit sequences of general-purpose and application-specific instructions.
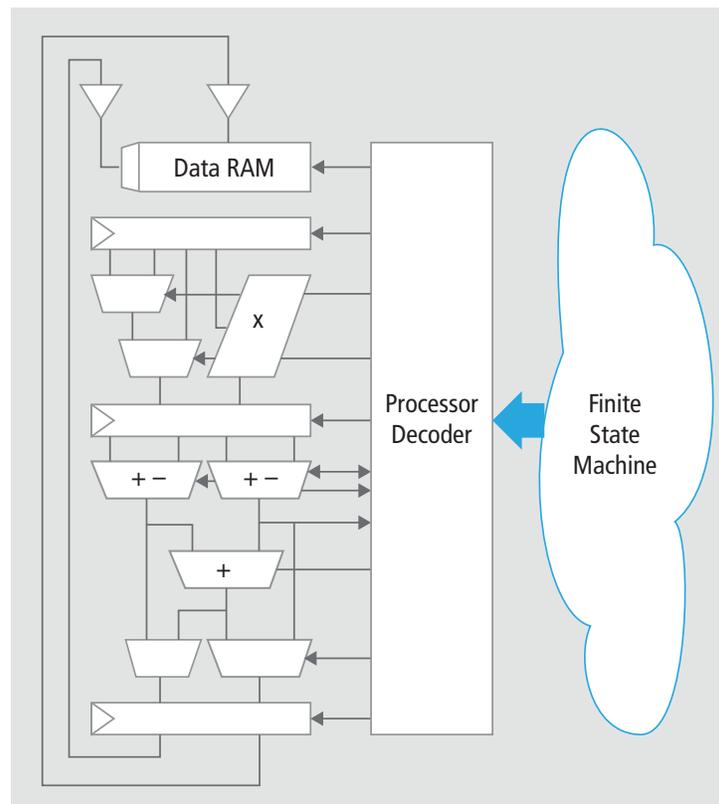
Figure 2: Programmable Hardware Function: Datapath + Processor + Software

Advantages of Using Processors Instead of RTL

Processors, such as Xtensa processors, enhanced with application-specific datapaths controlled by firmware-based FSMs have many advantages over hand-coded RTL accelerators:

• **Added flexibility**—Simply changing the FSM firmware changes a block's function.

• **Software-based development**—The ASIC design team can use fast, low-cost software tools to implement most chip features, adding more value to the silicon in less time.

• **Faster, more complete system modeling**—Even the fastest RTL logic simulator may not exceed a few system-simulation cycles per second, while firmware processor simulations run at hundreds of thousands or even millions of cycles per second on instruction-set simulators.

- **Unification of control and data**—No modern systems consist solely of hardwired logic—there is usually a processor running software. Moving functions that may have been previously implemented with hand-coded RTL functions into a processor removes the purely artificial separation between control and data processing and simplifies the system's design.

- **Time-to-market**—Using processors simplifies ASIC design, accelerates system modeling, and speeds hardware finalization, which in turn gets the product to market faster. After product introduction, firmware-based FSMs easily accommodate changes to standards because implementation details are not set in stone.

- **Designer productivity**—The engineering manpower needed for processors is greatly reduced when compared to developing and verifying custom RTL acceleration hardware. A processor-centric ASIC design approach permits graceful project-schedule recovery when bugs are discovered.

- **Reduced risk**—Customized processors reduce the risks associated with complex FSM development by replacing hard-to-design, hard-to-verify hardware FSMs with pre-designed, pre-verified processors.

- **Eliminate assembly coding**—Evolving to a more processor-centric design methodology moves much of the design definition to the C and C++programming languages instead of assembly coding. Application tasks defined in firmware that run on processors can be optimized, through customizable architectural extensions, to match the application performance of custom RTL accelerator blocks.

Fit the Processor to the Algorithm

ASIC developers can tailor each Xtensa processor for the target applications using special-purpose registers and register files of arbitrary width, specialized execution units, and wide data buses to reach an optimum processor design. These extensions can be created with just a few lines of specification.

The following examples show the performance improvements made possible by the Cadence Xtensa processor technology.

Example 1: Accelerating the Fast Fourier Transform (FFT) with the Xtensa Processor

This example contains information that is excerpted from the *Basic TIE Acceleration Techniques for DSP: FFT Example* Application Note. Refer to this application note for details. Also, an Xtensa Xplorer workspace is available to run all the discussed example cases.

Discrete Fourier transform (DFT) is used to transform digital samples in the time domain to the frequency domain. Fast Fourier transform (FFT) is one of the more efficient techniques used to calculate the DFT. Hence, the FFT is one of the most widely used DSP algorithms and is common in audio, video, and speech processing. This example shows a 256-point, fixed-point radix-2 FFT implementation on an Xtensa processor that will serve as a base implementation.

We focus on optimizations to the inner loop of the FFT algorithm that contains the radix-2 butterfly operation as shown in the following reference C code.

```
#define DESCALE(x, r) ((int) (((x) + (r)) >> 16))
const int round = (1 << (15))

void r2_fft_basic(int *data, int n, int *twiddles) {
    int i, d0r, d0i, d1r, d1i, r0r, r0i, r1r, r1i;
    int *p, *q, wr, wi;
    long long tr, ti;
    int m = n;
    int *t = twiddles;
    while (m > 1) { /* Outer loop (Decomposition into stages)*/
        for (i = 0; i < m; i += 2) { /* Middle loop */
            p = data + i;
            q = p;
            wr = *(t++);
            wi = *(t++);
            while (q < &data[n << 1]) { /* Inner loop */
                //Load Butterfly Inputs
                d0r = *(p + 0);
                d0i = *(p + 1);
```

```
            p += m;
            d1r = *(p + 0);
            d1i = *(p + 1);
            p += m;
            /Do Butterfly
            r0r = d0r + d1r; r0i = d0i + d1i;
            tr = d0r - d1r; ti = d0i - d1i;
            r1r = DESCALE(tr * wr, round) - DESCALE(ti * wi, round);
            r1i = DESCALE(tr * wi, round) + DESCALE(ti * wr, round);
            //Store Butterfly Outputs
            *(q + 0) = r0r;
            *(q + 1) = r0i;
            q += m;
            *(q + 0) = r1r;
            *(q + 1) = r1i;
            q += m;
                }
            }
            m >>= 1;
        }
    }
}
```

Several approaches are available to accelerate FFT performance on Xtensa processors, giving you the flexibility to choose a solution that matches your design constraints (e.g., performance, area, ease of use, generality). Acceleration for the FFT inner loop is illustrated using several TIE techniques that are simple and easy to implement.

Note that it is not the intent of this example to propose the highest performance or smallest solution. Rather, the intent is to demonstrate basic TIE techniques for accelerating any algorithm, not just FFT.

This example explores techniques to optimize performance or reduce area such as:

• **Complex number math TIE operations**—TIE gives you the ability to define a register file of any arbitrary width. In this example, we implement 24-bit fixed-point data values, and 48-bit TIE regfile for complex numbers consisting of 24-bit real and 24-bit imaginary fixed-point values. We define load/store/move operations and protos so that the Xtensa C Compiler (XCC) can appropriately manage the register file. We further define custom instructions to perform 48-bit complex number arithmetic, such as complex add, subtract, and multiplication operations.

• **Flexible Length Instruction eXtensions (FLIX)**—FLIX, a multi-issue VLIW with variable instruction length support, can be used to reduce cycle count. We can define wide instruction words with multiple operation slots. For our example, the FLIX1 configuration supports 8-byte instruction words with two or three operation slots and a single load/store unit.

• **Fusion**—Combine several operations into a single fused operation. For this example, we fused complex addition and store operations to support the top butterfly computation as well as the complex subtraction and multiplication operations to support the bottom butterfly computation. Fusing operations, standard ISA or TIE, reduce the execution cycles needed to perform computations.

• **Sharing hardware to reduce area**—This approach can be used for designs that have lower tolerance for area increases, and do not require such high performance. For example, rather than instantiating four separate multipliers in hardware to create the complex multiplier, the complex multiplication can be decomposed into four operations, each sharing a single hardware multiplier.

- **Single instruction, multiple data**—Perform the same operation (single instruction) on multiple data simultaneously. In this example, we define two-way SIMD operations for complex number addition, subtraction, and multiplication operations.

In Figure 3, each Xtensa processor configuration is tuned for performance or area criteria, as shown in Table 1.

Table 1: Xtensa Processor Configurations

| Xtensa Processor Configuration	Description	Notes
BASE_UNROLL	TIE	Adds custom complex number math operations
FLIX1_UNROLL	TIE + FLIX	FLIX—three operation slots Single load/store unit
BASE_FUSION	TIE + fusion	Fusion—multiple operations fused into a single instruction
FLIX1_FUSION	TIE + FLIX + fusion	FLIX—two operation slots
BASE_AREA	TIE	TIE—with discrete operations for the COMPLEX_MUL operations
FLIX1_AREA	TIE + FLIX	TIE –with discrete operations for the COMPLEX_MUL operations FLIX—three operation slots
BASE_SIMD	TIE + SIMD	SIMD—two-way SIMD operations for complex number addition, subtraction, and multiplication
FLIX1_SIMD	TIE + FLIX+ SIMD	FLIX—three operation slots
BASE_SIMD2	TIE + SIMD2	SIMD2—for reducing area, splits the two-way complex number multiplication containing eight multipliers into four operations that share two multipliers
FLIX1_SIMD2	TIE + FLIX + SIMD2	FLIX—three operation slots
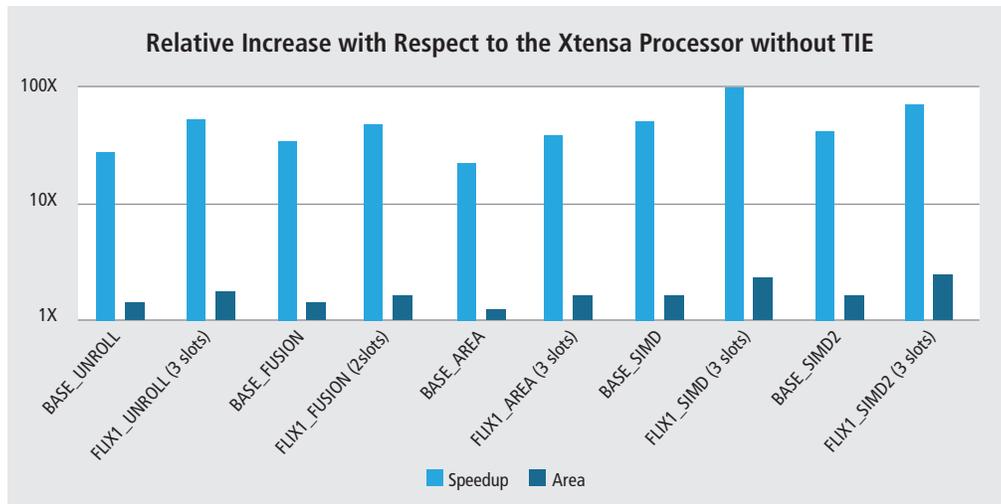
Figure 3: Relative Performance and Area Increases for Various TIE Configurations in 256-point Radix-2 FFT Implementations

Among the five approaches used to accelerate the radix-2 FFT, the highest performance solution results in 98X performance improvement compared to the base processor (with the MUL32 option), but the TIE gate count exceeds 72K gates. The smallest area solution results in 22X performance improvement, and only requires 12.8K gates.

These TIE techniques improve FFT performance by a factor of almost 100X compared to a conventional processor. While the radix-2 FFT algorithm is chosen to illustrate the use of TIE on a relatively simple DSP algorithm, these techniques can be used to accelerate any algorithm.

See the *Basic TIE Acceleration Techniques for DSP: FFT Example* Application Note for more details.

Example 2: Accelerating Computations for Image and Vision Processing with the Xtensa Processor

Image and vision processing requires a high number of computations to be performed quickly. In this example, we show how easily the Xtensa processor can accelerate histogram calculations with minimal added hardware, and then how the Xtensa processor, with additional customization, can accelerate bilateral filter processing.

Performing Histogram Calculations

A histogram is widely used in image processing, particularly for image analysis. Either the entire image or selected portions of it are analyzed to see the range and distribution of pixels. For an 8-bpp (bits per pixel) image, a pixel component (example: Luma) has 256 possible values, which means that the histogram can have 256 bins of gray values. After performing an image analysis on the entire image, you can see the region or level for most of the pixels. Based on histogram data, various other imaging techniques are utilized to perform further image enhancements. In an imaging application, histogram calculations are computed for the entire image, which requires full frame rates to be achieved—2, 16, or even 32 megapixels (MPs).

Calculating a histogram to compute 256-bin (level) calculations requires high computing throughput, for example using an Open Source Computer Vision (OpenCV) algorithm, which contains a library of programming functions for real-time computer vision. Today's camera systems are handling 16MP images at 30 frames per second (fps). Using a RISC processor takes 1.9 cycles/pixel, so for a 16MP image (assuming 8bps) this translates into 39.5fps for a RISC processor operating at 1.2GHz. A fully dedicated RISC processor just meets the demands for the histogram computations alone—it may not be able to handle greater processing demands.

To perform these specific histogram calculations, more processing acceleration can be added by designing application-specific TIE instructions that efficiently perform multiple operations within a single instruction. By adding new instructions to the Tensilica IVP-EP DSP to calculate the histogram, it performs 256 bin histogram calculations in 0.08 cycles/pixel. This is a speed up of ~24X and we added ~50K gates with a relative gate count increase of ~5% to the base DSP. Figure 4 shows that you can gain ~24X performance improvement for histogram calculations by adding TIE to the Xtensa IVP-EP DSP.
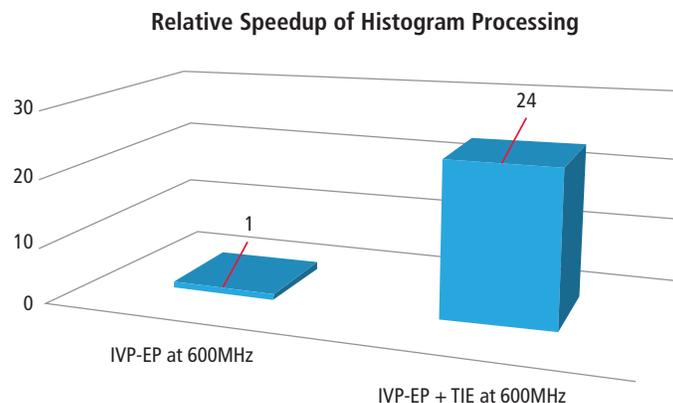
Relative Speedup of Histogram Processing

Figure 4: Tensilica IVP-EP DSP with TIE Relative Speedup for Histogram Processing

For example, the DSP with TIE running at 600MHz can process histograms at ~470fps, easily supporting a 16MP at 30fps sensor using only 6% of its processing horsepower. This leaves considerable processing power to perform additional image processing computations.

Note that you can gain additional performance improvements by adding more TIE instructions. You get to choose the optimal performance/size tradeoff for your application.

Implementing the Bilateral Filter

Image and vision processing is often based on spatial 2D filters or convolutions. A convolution is an integral of the product of one function (the filter kernel) as it is shifted over a second function (the image). For example, an image can be "blurred" by filtering it with a Gaussian filter kernel.

In image and vision processing, filters are commonly used to perform noise reduction on an image, or sharpen the image while preserving the original details in the image as much as possible. Usually these filters work on a local neighborhood of pixels such as 5x5, and the filter computation is repeated across the entire image.

One such example is a bilateral filter, a highly adaptive, non-linear, edge-preserving and noise-reducing smoothing filter used in image processing applications for de-noising, 3D depth filtering, texture editing and relighting, tone management, de-mosaicking, stylization, and optical-flow estimation.

Bilateral filtering requires a large number of computations, which result in unacceptably long execution times. The adaptive nature of the bilateral filter makes it challenging for traditional vector processors. Highly parallel processes can be used to efficiently process these large numbers of computations.

For example, for a 5x5 bilateral filter operation, the following operations must be performed for each pixel in the filter window on each color component of a pixel (e.g., R, G, B):

• Load 25 neighboring pixels

• Compute 25 difference values of the pixel with the current pixel

• Compute 25 absolute values of the above differences

• Shift the 25 absolute differences for quantization (to reduce the size of the filter coefficient table)

• Perform 25 table lookups for the kernel coefficient

• Compute 25 filter partial products (pixel * kernel coefficient)

• Sum the 25 partial products

• Sum the 25 kernel coefficients (normalization factor)

• Compute normalization (sum of partial products)/(sum of kernel coefficients)

The above steps render more than 600 total operations per pixel filtering with three pixel components (R,G,B). In a typical RISC processor, this results in more than 600 cycles per pixel filtering.

By optimizing the base Xtensa processor architecture using TIE to add more hardware, such as a 32-way SIMD, up to five-slot FLIX, and dual load/store units, this entire functional sequence can be executed in ~3.5 cycles per pixel. Reducing cycles per pixel from 600 to just 3.5 is a tremendous processing improvement from a typical RISC processor.

Table 2 shows pixel processing times assuming 8 bits/pixel. For VGA resolution, a typical RISC processor running at 1.2GHz processes ~6.5fps while the optimized Xtensa processor running at 600MHz (half the frequency) processes ~558fps—an 86X improvement. As the resolution increases, processing these types of filter operations in a reasonable amount of time becomes impossible on a RISC engine. However, the optimized Xtensa processor yields 172X pixel processing speedup compared to the typical RISC processor at the same clock frequency, irrespective of the resolution. Overall, the optimized Xtensa processor gives more processing power to meet the high computing demands of filtering operations while reducing the total energy used.

Looking at pixels per second, the optimized Xtensa processor at 600MHz processes over 171 million pixels per second, compared to a typical RISC processor at 1.2GHz that processes only about 2 million pixels per second. The optimized Xtensa processor executes vastly more computations at lower frequencies versus the RISC processor.

Table 2: Pixel Processing Speedup with the Optimized Xtensa Processor versus a General-Purpose RISC Processor

Resolution	Pixels	Optimized Xtensa Processor at 600MHz (fps)	RISC Processor at 1.2GHz (fps)	Optimized Xtensa Processor Speedup
VGA	(640 x 480)	558.04	6.51	86X
Full HD	(1920 x 1080)	82.67	0.97	86X
4K	(3840 x 2160)	20.67	0.24	86X

The optimizations in Table 2 were developed and incorporated into the Cadence Tensilica IVP-EP DSP, a high-performance embedded digital signal processor optimized for advanced image, video, and vision processing and analysis applications. The instruction set architecture and memory subsystem provides easy programmability in C and delivers a high sustained pixel processing performance.

The IVP-EP DSP is optimized for processing tasks characterized by both high pixel rates, exceeding 200M pixels per second through the major processing functions, as well as high computation rates, exceeding 1,000 operations on each pixel processed. Image, video, and vision processing algorithms are rapidly growing in complexity as new image methods and entirely new scene and object recognition applications are introduced. You can use this processor directly or you can further optimize it for your specific application.

Conclusion

Designers can accelerate the performance of many embedded algorithms by customizing Xtensa processors, adding precisely the computing resources (special-purpose registers, execution units, and wide data buses) required to achieve the desired algorithmic performance instead of designing these functions in hand-coded RTL acceleration hardware. Even Xtensa processors that use one of the various DSPs can be further customized for individual applications.

You can use this processor-centric design approach to ASIC design by utilizing profiling tools to analyze existing algorithm code to find the critical inner loops, a process well-understood by firmware-development teams. From these profiles, your ASIC design team can define new processor instructions and registers that accelerate critical loops, accelerating algorithm performance and producing a working function block much more quickly than designing and verifying an RTL accelerator block. Tuning a processor for your particular application instead of designing hand-coded hardware accelerators saves valuable design and verification time as well as adds an extra level of design flexibility because of the inherent programmability of the processor-centric design approach.

Additional Information

For more information on the unique abilities and features of the Cadence Tensilica Xtensa processors, see ip.cadence.com.

cadence®