Xtensa: A Configurable and Extensible Processor

System designers can optimize Xtensa for their embedded application by sizing and selecting features and adding new instructions. Xtensa provides an integrated solution that allows easy customization of both hardware and software. This process is simple, fast, and robust.

Ricardo E. Gonzalez
Tensilica, Inc.

Until a few years ago, processors were only sold as packaged individual ICs. The growing density of CMOS circuits, however, created an opportunity to incorporate the processor as part of a larger system on a chip. Initial processor designs for this market were based on the processor existing as a separate entity, and cores were handcrafted for each manufacturing process technology, resulting in costly and fixed solutions. Furthermore, it was not possible to modify these cores for the particular application, in much the same way that it was not possible to modify a stand-alone prepackaged processor.

Xtensa is a processor core designed with ease of integration, customization, and extension in mind. Unlike previous processors, Xtensa lets the system designer select and size only the features required for a given application. The configuration and generation process is straightforward and lets the designer define new system-specific instructions if preexisting features don't provide the required functionality. Furthermore, Xtensa fits easily into the standard ASIC design flow. Xtensa is fully synthesizeable, and designers can use the most popular physical-design tools during the place-and-route process.

Processor development

Application-specific processor development is an active area of research in the CAD, computer architecture, and VLSI design communities. Early attempts to add application-specific instructions to general-purpose computer engines relied on writable microcode.[1,2] These techniques dynamically augmented the base instruction set with application-specific instructions.

More recent research focuses on automatic instruction set design[3,4] or on reconfigurable, also called retargetable, processors.[5] These groups, however, try to solve slightly different problems than those addressed by Xtensa. Automatic instruction set design systematically analyzes a benchmark program to derive an entirely new instruction set for a given microarchitecture. Our group—here, referred to as "we"—focuses on how to generate a high-performance and low-power implementation of a given microarchitecture with application-specific extensions. In this respect, automatic instruction set design is a good complement to our work. Once the instruction set additions are derived automatically by analyzing the benchmark program, they can be given to the Xtensa processor generator to obtain a high-performance, low-power implementation.

Reconfigurable or retargetable processors couple a general-purpose computer engine with various amounts of hardware-programmable logic. In the extreme, the entire processor is implemented using hardware-programmable logic. This technique, however, is limited by the large difference in operating frequency between programmable and nonprogrammable logic. Processors implemented entirely using programmable logic operate an order of magnitude slower than nonconfigurable processors implemented in a comparable process technology.

Razdan and Smith present an interesting compromise.[5] Their approach couples a custom-designed high-performance processor with small amounts of hardware-programmable logic. Their system uses compiler-generated information to dynamically reconfigure a small amount of hardware-programmable logic to implement new application-specific functional units. This technique also has limitations due to the disparity in operating frequency of programmable and nonprogrammable logic. Thus, the new functional units must be extremely simple or be deeply pipelined.

Our approach is similar to that taken by Razdan and Smith, however we don't attempt to dynamically reconfigure the system. The Tensilica processor generator adds the application-specific functionality at the time the hardware is designed. Thus, the extensions are implemented in the same logic family as the rest of the processor. This eliminates the disadvantages of using programmable logic for implementing the extensions, but precludes modification of the extensions for different applications.

Due to a lack of automated tools, designers incorporated application-specific functionality in CPUs by adding specialized coprocessors.[6,7] This approach produces communication overhead between the CPU and the coprocessor, making system design more arduous.

Recently, with the advent of synthesizeable processors, some groups have proposed manual modification of the register-transfer level (RTL) description of the processor and the software development tools.[8] This approach is tedious and error prone. Furthermore, the extensions are only applicable to one implementation. If users want to add similar extensions to a future implementation of the same processor, they must modify the RTL again.

Our research differs from previous studies because we use a high-level language to express processor extension. This language, called Tensilica Instruction Extension (TIE), expresses the semantics and encoding of instructions. TIE can add new functionality to the RTL description and automatically extend the software tools. This lets the system developer code applications in a high-level language, such as C or C++. TIE imposes restrictions on functions that designers can describe, which greatly simplify verification of the processor and extensions. Because the extensions become an integral part of the processor, there is no communication overhead.

Synthesizeable processors

Traditionally, processors are custom designed. If designers employ logic synthesis, it is only to generate control modules they feel are not timing critical. Designers take great care in controlling the layout to avoid parasitics and capacitive coupling between nodes (allowing the use of dynamic circuits). Furthermore, custom design allows the use of sophisticated circuit structures including content addressable memories and specialized RAMs. These circuit structures can efficiently implement particular microarchitectural features, such as translation look-aside buffers, address lookup, and fast RAM access. Custom circuit design can result in very high operating frequencies, as evidenced by recent announcements from Intel, AMD, IBM, and others. Well-designed custom processors can operate at 700 MHz[9] and will reach 1 GHz in the near future. However, few custom-designed circuits can take full advantage of this potential. Most custom circuits are compromised by porting, short development time, or inappropriate design methods. This results in custom processors that often operate at a fraction of this maximum frequency.

Synthesizeable processors cannot match the raw frequency potential of well-designed custom processors. We found that the maximum operating frequency of the design is restricted by the limited functionality of standard-cell libraries, the higher parasitics associated with circuits generated using automated tools, and the larger clocking overhead. However, configurability and extensibility more than

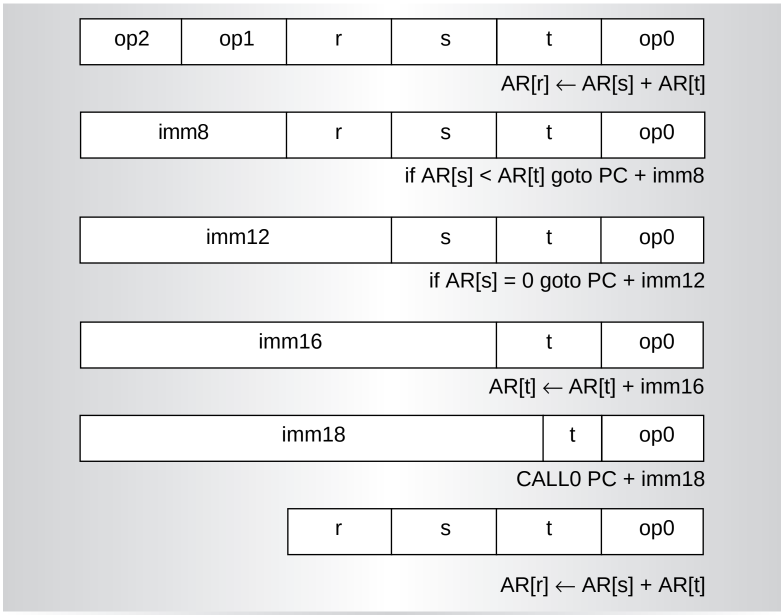

Figure 1. Xtensa instruction formats (little-endian).

compensate for this difference in maximum operating frequency.

Although they may have the highest potential operating frequency, custom-designed processors are not well suited for embedded system-on-a-chip applications for three reasons. First, they use different CAD tools than the rest of the system. ASICs are designed using logic synthesis and automated place-and-route tools. Incorporating a custom-designed processor into this flow is problematic. Next, custom-designed processors must be manufactured using a particular process and foundry, and porting the processor to a new foundry or process usually requires several months. Furthermore, the processor rarely takes full advantage of the new process. Completely reoptimizing the processor for this new manufacturing process isn't feasible since it would take nearly as much time as the original design. This is independent of whether the new process is in the same foundry or not. Thus, production of a custom-designed processor cannot occur in a new manufacturing process until 6 to 12 months after the process is available. Finally, it's not feasible, from a time-to-market standpoint, to modify a custom-designed processor to better match the application.

Synthesizeable processors, conversely, do not suffer from these limitations and are therefore easier to integrate into a large ASIC. They naturally match the ASIC design flow and simply become one more integrated block. Synthesizeable processors are not tied to a particular foundry. In addition, they can take advantage of manufacturing process improvements immediately because there is no lead time to port the design. A synthesizeable processor can be quickly manufactured with a different foundry or in a more advanced process. Moreover, because the circuit implementation is automatically generated, designers can map the same RTL description multiple times with different optimization goals, resulting in different implementations. The same RTL description can produce separate implementations that are high performance, low power, and low cost, by simply changing the synthesis constraints and the target library. The biggest benefit of synthesizeability, however, is that it enables configuration and extension by the designer.

We expect the number of synthesizeable processors, such as Xtensa, to increase in the coming years as designers realize the benefits of using logic synthesis, despite the theoretical advantage of operating frequency in custom-designed processors.

Overview of Xtensa

Since synthesizeability has several advantages for embedded processors aimed at system-on-a-chip designs, we designed Xtensa to exploit these advantages. The Xtensa instruction set architecture (ISA) and the hardware implementation also streamline extensibility and configurability.

Instruction set architecture. Xtensa's ISA enables configurability, minimizes code size, reduces power dissipation, and maximizes performance.

The Xtensa ISA consists of a base set of instructions, which exist in all Xtensa implementations, plus a set of configurable options. The designer can choose, for example, to include a 16-bit multiply-accumulate option if it is beneficial to the application. The base ISA defines approximately 80 instructions and is a superset of traditional 32-bit RISC instruction sets.[10]

The architecture achieves smaller code size through the use of denser encoding and register windows. The ISA defines 24- and 16-

bit instruction formats, as opposed to 32-bit formats found in traditional RISC instruction sets. The Xtensa architecture provides a rich set of operations despite the smaller instruction size. These sophisticated instructions, such as single-cycle compare and branch, enable higher code density and improve performance. Figure 1 shows the six instruction formats available. We designed the instruction encoding to let the compiler use the smaller instructions for the most common operations, further reducing code size. Register windows reduce code size by eliminating the register saves and restores required at the entry and exit of every subroutine.

Some 32-bit RISC ISAs have 16-bit instructions to reduce code size. These ISA additions accomplish that goal but usually at the expense of processor performance. These narrow instructions, for example, have severely limited immediate sizes that require multi-instruction sequences to perform common operations. Furthermore, these narrow instructions cannot access all of the processor's registers. Unlike Xtensa, these architectures rely on a dynamic mode bit to determine instruction size. And, to simplify the hardware implementation, the architectures allow changes to the mode bit only at procedure call boundaries. Thus, the programmer or compiler must trade off code size and performance at a relatively coarse level in the program.

The size of Xtensa instructions is encoded in the instruction, enabling 24- and 16-bit instructions to freely intermix at a fine granularity. The 16-bit instructions are a subset of the 24-bit instructions. Thus, the compiler optimization to reduce code size is trivial: replacing 24-bit instructions with their 16-bit equivalent. The compiler can reduce code size without sacrificing performance.

Table 1 compares code size of the Xtensa and MIPS architectures. We compiled the benchmarks for both architectures using the GCC C compiler. Code size for Xtensa is roughly half that of MIPS.

The ISA achieves high performance by providing richer instructions, enabling high-speed implementations, and reducing register save and restore overhead in subroutines. The architecture also supports zero-overhead loop instructions that can significantly speed up small computation-intensive kernels. As evident in Figure 1, the placement of the register specifiers is the same for all instruction formats—the same bits of the instruction register are always used to access the register file. This placement allows for simpler and faster implementations.

Table 1. Comparison of code size for Xtensa and MIPS.

Benchmark	Description	Xtensa (Kbytes)	MIPS (Kbytes)
Jpeg-6	Image compression and decompression	107	214
Li	SPEC benchmark	30	71
Soft modem	Modem codec	763	1,230
VxWorks	Real-time operating system	339	607

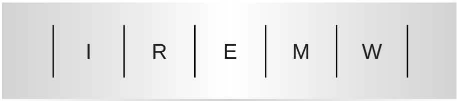

Figure 2. The Xtensa pipeline.

Hardware implementation. We built the first implementation of Xtensa around a traditional RISC five-stage pipeline (see Figure 2), with a 32-bit address space.[10]

The processor accesses the instruction cache and tags in the first half of the I stage and computes the cache hit or miss signal in the second half. At the same time, the machine aligns the data read from the instruction cache to present a valid instruction to the fetch unit. Alignment is necessary because instructions are odd sizes (2 and 3 bytes). Like traditional RISC processors, the instruction is decoded, and the register file is accessed in the R stage. In the E stage, the machine computes the effective address for loads and stores, and executes ALU instructions. During the E stage, the machine also determines whether a conditional branch is taken and, if so, updates the PC. This results in a two-cycle bubble for taken branches.

Xtensa presents the effective address to the data cache and data tags at the end of the E stage. For loads, the processor accesses the data cache in the first half of the M stage and computes the cache hit signal in the second half. Thus, loads have a latency of two cycles. Finally, instructions that update the register file do so in the W stage. The machine implements zero-overhead loops by comparing the

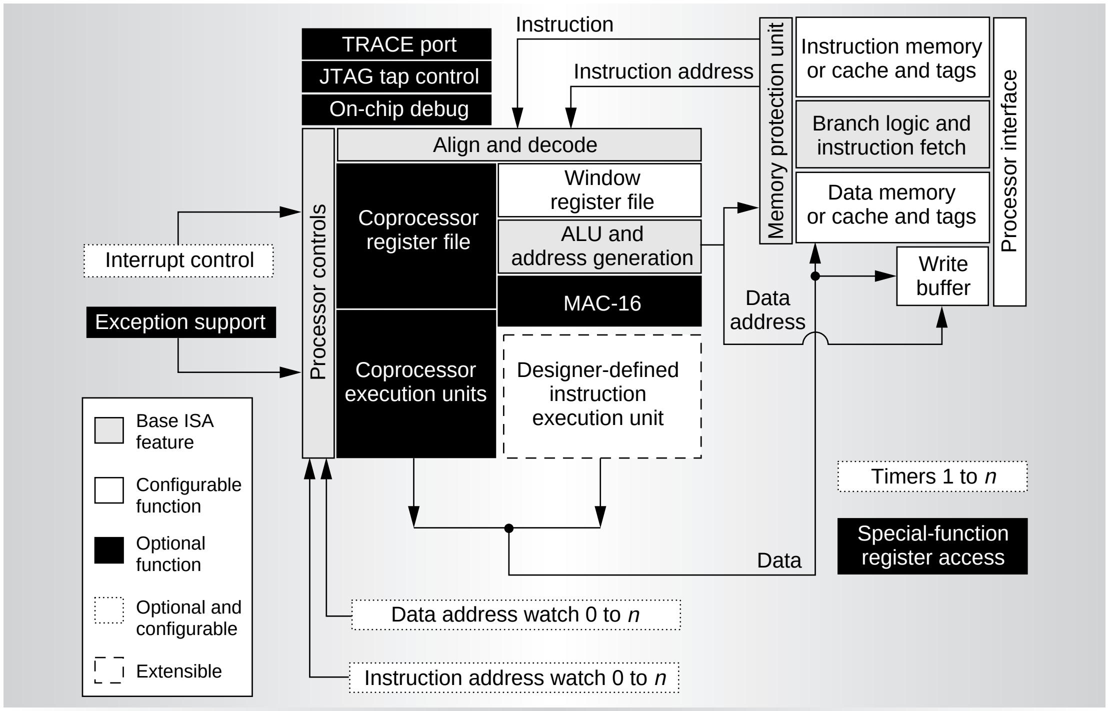
TRACE port
JTAG tap control
On-chip debug
Instruction
Instruction address
Memory protection unit
Instruction memory or cache and tags
Branch logic and instruction fetch
Data memory or cache and tags
Write buffer
Processor interface
Align and decode
Processor controls
Coprocessor register file
Window register file
ALU and address generation
MAC-16
Coprocessor execution units
Designer-defined instruction execution unit
Interrupt control
Exception support
Data address
Data
Timers 1 to n
Special-function register access
Data address watch 0 to n
Instruction address watch 0 to n
Base ISA feature
Configurable function
Optional function
Optional and configurable
Extensible

Figure 3. A high-level block diagram of Xtensa.

Table 2. Examples of Xtensa configuration parameters.

Parameter	Legal values
Instruction cache size	1, 2, 4, 8, and 16 Kbytes
Data cache size	1, 2, 4, 8, and 16 Kbytes
Data RAM size	1, 2, 4, 8, and 16 Kbytes
Size of windowed register file	32, 64 registers
External bus width	32, 64, 128 bits
Number of interrupts	0-32
Interrupt levels	0-6
Timers	0-3
Memory order	Big-endian, little-endian

PC in the I stage with the end-of-the-loop pointer. If a match occurs, the PC is set to point to the start of the loop and causes a decrement of the loop count.

Many other characteristics of the processor, however, are configurable. The configurability and extensibility of the implementation matches those of the architecture. Figure 3 shows a high-level block diagram of Xtensa. The base ISA features correspond to roughly 80 instructions. Configurable options show designer choices, such as how many physical registers to include in the implementation, or the size of the instruction and data caches. Optional features are selections the designer can make, such as whether to include a 16-bit multiply-accumulate functional unit. Optional and configurable functions let the designer select whether to include data watchpoint registers and, if so, how many. Another configurable option is the designer-defined functional unit.

Table 2 shows some of the configuration parameters and associated legal values available in the first Xtensa implementation. Unlike conventional processors, Xtensa gives designers a choice regarding the functionality of the processor.

The speed, power, and area of an Xtensa core depend on many factors, such as the goals set by the designer during the synthesis process and the target library used. Table 3 shows the speed, size, and power dissipation—as reported by the synthesis tool—of two cores. One core is optimized for speed and the other is optimized for minimum area. To validate the values predicted by the synthesis tools, we developed a test chip using an Xtensa core. The test chip, manufactured in Taiwan Semiconductor Manufacturing Company's 0.25-micron CMOS process, is fully functional and operated at slightly over 200 MHz in the lab.

The synthesis tool had predicted worst-case (across temperature, voltage, and process) operating speed of 140 MHz.

Configuration. The configuration process begins by accessing the Tensilica processor generator Web page at http://www.tensilica.com. (It is also possible to install the generator locally and access the Web page over a local Intranet.) Here, using a standard browser, the designer can select and size the desired features. The site's configuration page gives the designer instant feedback on whether a particular choice will affect the speed, power, or area of the core. The user interface warns the designer of conflicting options or requirements for a particular option.

The designer starts the generation process by clicking a button. The generation process produces the processor's configured RTL description and its configured software development tools. The software tools consist of an ANSI C/C++ compiler, linker, assembler, debugger, code profiler, and instruction set simulator.

At the beginning of the generation process, the Tensilica processor generator builds a database of all configuration options. The generator then processes each file in the source code of the software tools and the RTL model. Each file can contain, if necessary, configuration information in a metalanguage. This metalanguage describes the transformations required to generate a configuration-specific description. Once the configuration step is complete, the generator builds the software tools to produce the required executables.

The software tools are built on top of the industry-standard GNU tools. The generation process takes one hour to complete. After the process is complete, the designer can download and install the RTL and software development tools. At this point, the designer can either compile an application and measure the performance using the instruction set simulator or start the hardware synthesis using the RTL description.

The software development tools include an optimizing C compiler enabling application development in a high-level language. The instruction set simulator and code profiler help the designer quickly identify bottlenecks in application performance. Optionally, the designer can recode the application to work around these bottlenecks or add new instructions to the processor designed to optimize this particular application.

The designer can map the RTL description to a gate-level netlist using industry-standard synthesis tools. Included with the RTL description are a set of synthesis scripts that help automate this process. These scripts let designers quickly obtain a fully optimized gate-level netlist of Xtensa. Tensilica also provides a set of scripts to automate the place-and-route process. It is common for new users to place and route Xtensa within a day or two of downloading the configured RTL.

Table 3. Xtensa speed, power, and area (0.25 micron, worst-case conditions).

Optimization	Core area (mm^2)	Gates (Kbytes)	Power (mW/MHz)	Speed (MHz)
Area	1.1	25	0.8	100
Speed	1.5	35	1.0	140

Extension via TIE

Tensilica Instruction Extension (TIE) is a language that lets designers incorporate application-specific functionality in the processor by adding new instructions.

It is not practical to expect the processor architect to foresee all possible applications of the core and include suitable instructions for these applications. The processor architect's job is, in fact, to decide which applications are common enough to warrant some level of support through dedicated instructions. In recent years, for example, processor architectures have added single-instruction, multiple-data instructions to better support multimedia applications. Many other applications can benefit from dedicated hardware. Some applications, for example, require bit manipulations, which are awkward with traditional architectures. Other applications require specialized shift or rotate operations. Yet another class of applications may require a different form of arithmetic. The number of opportunities to add specialized instructions is limitless.

TIE lets the designer specify the mnemonic, the encoding, and the semantics of single-cycle instructions. It also lets the designer declare new processor state (we also refer to

```
// define a new opcode for byteswap
opcode   BYTESWAP  op2=4'b0000       CUST0

// declare state SWAP and ACCUM
state  SWAP          1
state  ACCUM         40

// map ACCUM and SWAP to user register file entries
user_register         0      ACCUM[31:0]
user_register         1      {SWAP, ACCUM[39:32]}

// define a new instruction class
iclass   bs  {BYTESWAP}   { out  arr , in  ars } { in  SWAP,
     inout  ACCUM}

// semantic definition of byteswap
semantic   bs {BYTESWAP} {
  wire  [31:0] ars_swapped =
     {ars [7:0], ars[15:8], ars  [23:16], ars[31:24]};
  assign  arr = SWAP ? ars_swapped : ars ;
  assign  ACCUM = {ACCUM[39:30] + arr[31:24],
                   ACCUM[29:20] + arr[23:16],
                   ACCUM[19:10] + arr[15:8],
                   ACCUM[ 9: 0] + arr[7:0]};
}
```

Figure 4. A simple TIE example.

this as a state register). New TIE instructions can access either the built-in processor state (defined by the Xtensa ISA) or a new user state.

The designer can group instructions that have similar operands into classes. A class can contain one or more instructions. The behavior, or semantics, of an instruction are described using a subset of the Verilog hardware description language. To produce a result, the semantic block simply assigns a value to the output.

User state registers can hold intermediate values or control information. The designer can optionally declare state registers as part of a user-defined register file. This declaration does two things. It lets the TIE compiler automatically generate a library that a binary distribution of an RTOS can use to save and restore processor state. It also makes the user state register accessible via predefined instructions like a conventional register file.

An example of a simple TIE description is shown in Figure 4. In this example, TIE keywords are shown in bold, TIE predefined names are underlined, and comments are shown in italics.

In Figure 4's example, the opcode declaration defines the mnemonic and encoding of a new instruction called BYTESWAP. The state declarations define two new state registers: a single-bit control register (SWAP), and a 40-bit accumulator (ACCUM). The user_register declaration adds these two user state registers to the user-defined register file. The iclass statement defines a new instruction class with the single instruction (BYTESWAP). Instructions in this class use arr as an output, ars as an input, the SWAP user state register as an input, and the ACCUM user state register as both an input and an output. The behavior of the new instruction (BYTESWAP) is described in the semantic block. The instruction conditionally reverses the byte order of the input operand and accumulates the values of the different byte lanes.

TIE is easy to write because it abstracts the implementation details. The semantics of the instruction are described as if the instruction consisted of only combinatorial logic. TIE is independent of the processor's pipeline. Designers do not have to implement the logic required for bypass detection, interlocks, and so on. Because designers can think of new functionality as if it were combinatorial logic, it is unnecessary for them to allocate the logic to particular pipeline stages. Furthermore, designers can use the same TIE description with multiple Xtensa implementations. The abstraction of implementation details makes TIE a very powerful language; a few lines of TIE code can add significant functionality to the processor.

The TIE compiler automatically adds the new instructions to the RTL and the software tools. It translates the instruction semantics to the appropriate hardware description language and produces the appropriate pipeline control signals. It automatically generates the bypass control, interlock detection, and immediate-generation logic required by the instructions.

The generated hardware fits seamlessly into the Xtensa pipeline, as shown in Figure 5. In the figure, white boxes represent combinational logic in the processor. Boxes with dashed lines represent flip-flops, and bold boxes represent combinational logic added by the TIE compiler. Figure 5 shows two stages of the traditional RISC five-stage pipeline. In the R stage, the machine reads the contents of the register file, decodes the instruction, and detects bypass conditions and interlocks. In the beginning of the E stage the machine dispatches the operands to one of the functional units, and at the end of the E stage it selects one of the results from the functional units. The generated hardware is indistinguishable from the native functional units and state registers. The TIE compiler extends the bypass detection and interlock logic to recognize the new instructions.

The TIE compiler also automatically extends the software tools. It adds the new instructions as intrinsics to the C and C++ compiler, letting designers develop their application in a high-level language. The TIE instructions go through the register allocation and optimization stages of the compilation. Thus, the software developer can use C variables when calling the intrinsic functions.

The TIE compiler also translates the semantics of the new instructions to a native C implementation, which the designer can use to verify the functionality of the instructions. This lets the designer debug the instructions using a desktop workstation and run hundreds of thousands of instructions per second rather than using RTL simulations and run only hundreds of instructions per second.

Figure 6 shows a typical TIE development cycle. Designers start by developing an application in C or C++. They then compile the application to the Xtensa architecture and use the instruction set simulator to measure application performance. If performance is not acceptable, the designer can use the code profiler to identify bottlenecks and potential new instructions. They would then implement these new instructions in TIE and use the native C description to debug the TIE instructions. Once the correct operation of the instructions is confirmed, designers can recompile the application for Xtensa and again measure application performance using the instruction set simulator. If acceptable, designers can proceed to synthesize and build the processor.

The profile, implement, and debug cycle usually takes from one to three hours, which lets the designer explore several different solutions in a single day. The designer is thus better able to quickly find a good solution to the problem.

Adding extensions via TIE does impose some limitations. For example, the added logic must fit into the core pipeline, and there are some restrictions on the placement of source and result operands in the instruction encoding. These restrictions, however, are what make TIE safe and easy to use.

As mentioned previously, designers have added application-specific functionality to CPUs either by including additional coprocessors or by manual modification of the proces-

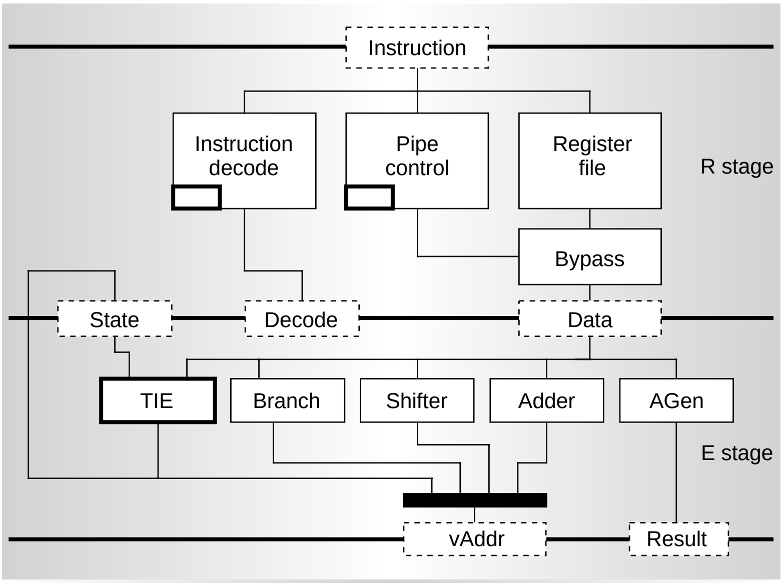

Figure 5. TIE hardware generation.

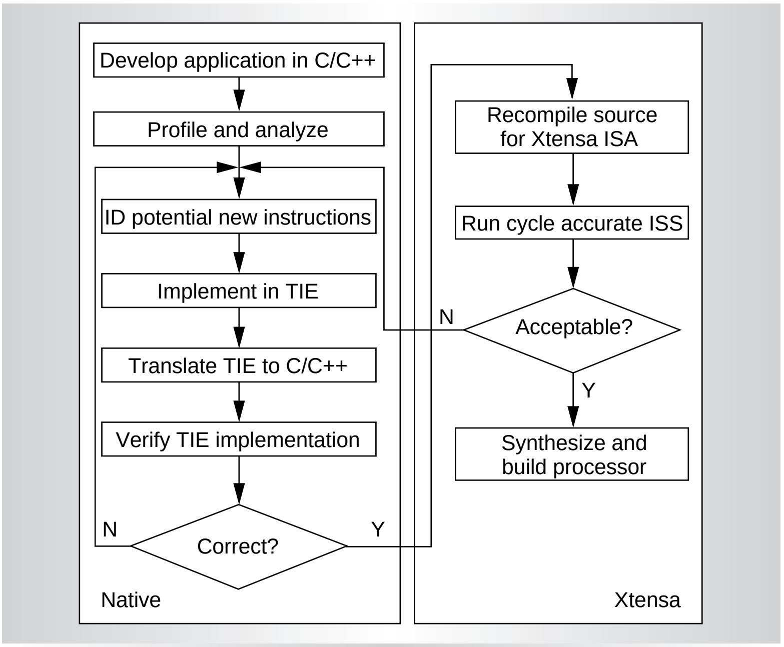
Develop application in C/C++
Profile and analyze
ID potential new instructions
Implement in TIE
Translate TIE to C/C++
Verify TIE implementation
N
Correct?
Y
Native
Recompile source for Xtensa ISA
Run cycle accurate ISS
N
Acceptable?
Y
Synthesize and build processor
Xtensa

Figure 6. A typical TIE development cycle.

sor's RTL description. Adding extension by way of TIE has several advantages when compared with these traditional methods. If extensions are added by way of a specialized coprocessor, there is often overhead in communication between the core and the coprocessor. If an extension is done with TIE, the new hardware is seamlessly integrated into the pipeline. In addition, TIE lets the designer accomplish complicated control sequences using the instruction stream. In a coprocessor, the designer must implement the control sequence as a finite-state machine, which is impossible to change once manufactured. With TIE, it may be possible in some cases to change the instruction sequencing to work around a problem. TIE is also easier to verify due to the faster simulation times (four or five orders of magnitude faster than with RTL).

Adding extensions by modifying the processor's RTL requires the designer to become intimately familiar with the processor's design. Once the designer modifies the RTL, it is then necessary to modify some of the software development tools. The designer must add new instructions to the assembler, the debugger, and perhaps the linker. To enable the designer to develop the application in a high-level language, it is also necessary to modify the compiler. Conversely, if the extension is coded in TIE, the hardware and software are configured together automatically. Again, it is much easier to verify the TIE implementation compared to the RTL implementation due to the faster simulation time. Also, manual modification of the processor's RTL description could take several months.

TIE example: DES. To demonstrate the potential of TIE, we extended Xtensa to improve the performance of the Data Encryption Standard (DES)—a popular encryption and decryption algorithm often used for secure Internet communication. We chose DES for two reasons: its growing popularity in embedded applications that require secure Internet transactions, and the relatively poor encryption and decryption performance of general-purpose processors.

A simple DES modification, known as Triple-DES, extends the key to 168 bits by iterating the DES algorithm three times with three different keys. Triple-DES has been specified as an encryption algorithm for both the secure shell tools[11] and the Internet Protocol for Security.[12] Both of these applications require high-speed encryption and decryption and are implemented as part of many of today's interesting embedded systems.

The DES algorithm requires extensive bit permutations, which are difficult to implement efficiently in software. However, designers can efficiently implement these permutations in hardware, since each corresponds to a simple renaming of the wires. The algorithm also specifies rotation on 28-bit boundaries. Even if the processor has a rotate instruction, it often is not usable since it most likely rotates on 32-bit boundaries. Finally, the algorithm requires bit packing and unpacking, and table lookups. These operations are slow in software but easy to implement with hardware. We modified the Xtensa processor to include special instructions to speed up these operations.

We added enhancements to the processor following the steps in Figure 6. We started by developing a reference implementation of DES based on the source code for secure shell tools (see ftp://ftp.cs.hut.fi/pub/ssh). Based on runtime profile information, we then defined four new instructions and reimplemented the application to use these instructions. We verified the implementation of the TIE instructions by comparing the output of the modified application, which uses the TIE-generated C description of the instructions, with the results of the reference C implementation.

In addition to four new instructions, we also added three new state registers to the processor. The registers hold intermediate values during the encryption and decryption process. Of the four new instructions, one performs the encryption and decryption step using values in the state registers. The other three instructions transfer data to (and from) the processor registers from (and to) the state registers, while concurrently permuting the data values.

When compiled for the Xtensa architecture, the new application required only 154 bytes of object code and no static or dynamic data storage. Thus, the original implementation required 36 times more memory than this implementation.

Figure 7 shows the speedup of the DES-enhanced Xtensa core compared to an unmodified Xtensa core. The X-axis shows the block size used for encryption and decryption. The original DES implementation gains much of its speed by precomputing large tables of values from a fixed key, making key changes very expensive. Thus small blocks can attain speedup by a greater factor than large blocks (where key changes are less frequent). The modified Xtensa can encrypt and decrypt data at the rate of 377 Mbytes/s. The hardware cost of the TIE instructions is roughly 4,500 equivalent (NAND2) gates (measured in a 0.25-micron process technology). The reduced storage requirements of the application offset this hardware cost. In addition, the new TIE instructions did not increase the cycle time of the machine.

DES is only one of the applications that can benefit from specialized hardware. Figure 8 shows similar speedup plots for other applications common in embedded processors, such as image compression, Viterbi decoding, motion estimation, and FIR filtering. In all cases, the cost of the additional hardware is modest while the performance gains are large (from 4 to 53 times).

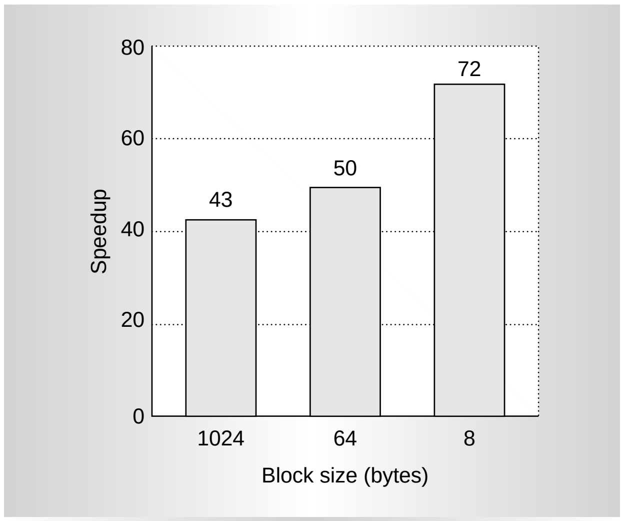

Figure 7. DES speedup using TIE.

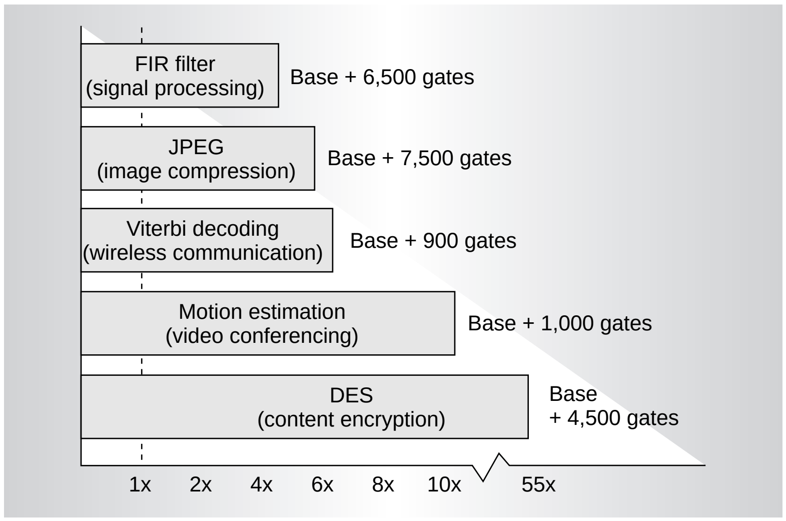

Figure 8. TIE-based speedup.

Synthesizeable processors are a reality. They are competitive with traditional offerings in speed, power, and area; they are portable; and they can quickly and easily migrate to new processes.

Configurable and extensible processors are here. The key to configurability is the integration of hardware and software, letting the designer select and size only the features needed without concern for software development tool availability. Configurable and extensible processors enable designers to quickly find better solutions to their problems. MICRO

Acknowledgments

I would like to thank the entire engineering staff at Tensilica. This article was made possible by their work. I also thank Rick Rudell for the development of the TIE code for DES and Monica Lam for her valuable comments and discussions.

References

1. A. Abd-alla and D. Kartlgaard, "Heuristic Synthesis of Microprogrammed Computer Architectures," *IEEE Trans. on Computers*, Vol. 23, No. 8, Aug. 1974, pp. 802-807.
2. P. Liu and F. Mowle, "Techniques of Program Execution with a Writable Control Memory," *IEEE Trans. on Computers*, Vol. 27, No. 9, Sept. 1978, pp. 816-827.
3. F. Haney, *Using a Computer to Design Computer Instruction Sets*, doctoral dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Penn., 1968.
4. B. Holmer and A. Despain, "Viewing Instruction Set Design as an Optimization Problem," *Proc. 24th Int'l. Symp. on Microarchitecture*, IEEE Computer Society, Los Alamitos, Calif., 1991, pp. 172-180.

5. R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," *Proc. Micro-27*, IEEE Computer Society, 1994, pp. 172-180.
6. O. Nishii et al., "A 200-MHz 1.2-W 1.4-GFLOPS Microprocessor with Graphics Unit," *Proc. IEEE Int'l Solid-State Circuits Conf.*, Vol. 41, IEEE Press, Piscataway, N.J., 1998, pp. 288-289.
7. S. Santhanam et al., "A Low-Cost 300-MHz RISC CPU with Attached Media Processor," *Proc. IEEE Int'l Solid-State Circuits Conf.*, Vol. 41, IEEE Press, 1998, pp. 298-299.
8. http://www.arccores.com.
9. S. Hesley et al., "A 7th-generation x86 Microprocessor," *Proc. IEEE Int'l. Solid-State Circuits Conf.*, Vol. 42, IEEE Press, 1999, pp. 182-183.
10. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, Calif., 1990.
11. T. Ylonen et al., *SSH Protocol Architecture*, Internet-Draft, 1998; http://freenic.net/drafts/drafts-i-j/draft-ietf-secsh-architecture-02.txt.
12. S. Kent and R. Atkinson, *Security Architecture for the Internet Protocol*, RFC 2401, Nov. 1998; ftp://ftp.isi.edu/in-notes/rfc2401.txt.

Ricardo E. Gonzalez is a member of the technical staff at Tensilica, Inc., where he is responsible for the development of high-performance and low-power configurable processor cores. Before joining Tensilica, he was a member of the technical staff at Intel's Microcomputer Research Lab. While at Intel he explored new architectural ideas for very high-performance processors. He received his BS, MS, and PhD from Stanford University. His interests include VLSI design, low-power and high-performance circuits, computer architecture, and CAD tools.

Direct questions about this article to the author at Tensilica, Inc., 3255-6 Scott Blvd., Santa Clara, CA 95054; ricardog@tensilica.com.