Hardware/Software Instruction Set Configurability for System-on-Chip Processors

Albert Wang
awang@tensilica.com

Earl Killian
earl@tensilica.com

Dror Maydan
maydan@tensilica.com

Chris Rowen
rowen@tensilica.com

Tensilica, Inc.
3255-6 Scott Blvd.
Santa Clara, CA 95054
+1 408 986 8000

ABSTRACT

New application-focused system-on-chip platforms motivate new application-specific processors. Configurable and extensible processor architectures offer the efficiency of tuned logic solutions with the flexibility of standard high-level programming methodology. Automated extension of processor function units and the associated software environment – compilers, debuggers, simulators and real-time operating systems – satisfies these needs. At the same time, designing at the level of software and instruction set architecture significantly shortens the design cycle and reduces verification effort and risk. This paper describes the key dimensions of extensibility within the processor architecture, the instruction set extension description language and the means of automatically extending the software environment from that description. It also describes two groups of benchmarks, EEMBC's Consumer and Telecommunications suites, that show 20 to 40 times acceleration of a broad set of algorithms through application-specific instruction set extension, relative to high performance RISC processors.

1. WHY CONFIGURE PROCESSORS?

Two major shifts – one technical, one economic – are changing the design of electronic systems. First, continuing growth in silicon chip capability is rapidly reducing the number of chips in a typical system, and magnifying the size, performance and power benefits of system-on-chip integration. Second, many of the fastest-growing electronics products demand ever-better cost, bandwidth, battery life, and software functionality. These systems – network routers, MP3 players, cell-phones, home gateways, PDAs, and many others – require both full programmability (to manage complexity and rapidly evolving requirements) and high silicon efficiency (for superior application performance per watt, per dollar and per mm^2). Application-specific processor cores promise such a combination of full software flexibility with high efficiency

The demand for application-specific processors creates a paradox for modern system design: how do architects develop new processors that combine the key benefits of generic programmable chips – longevity, development costs amortized over large volume, adaptability to changing market requirements – without taking too much development time or expense. If the cost of fashioning new optimized processors could be radically reduced, then a much broader array of highly refined processor cores could be used in system-on-chip designs.

Tensilica enables rapid design of highly efficient processor cores by providing a base architecture, a lean core implementation, and an automated method to seamlessly extend the processor hardware and software to fit each system's application requirements. Processors extended by this methodology close the performance gap between high-overhead general-purpose programmable processors and efficient, specialized hardware-only solutions based on hardwired-datapath-plus-state-machine logic functions [1]. This methodology also closes the design gap between the rapid, exponential growth of silicon capacity and the slower growth in designer productivity [2]. This paper outlines the capabilities of Tensilica's Xtensa processor generator [3], including the Tensilica Instruction Extension (TIE) methodology and demonstrates a resolution to the paradox.

2. WHAT'S THE RIGHT ARCHITECTURE?

But what is the right new architecture for extended processors? What instructions should we add? There is no universal extension, or even one for each application class. System designers may already know the answer for their own problem area. Good candidate instructions can be found in the datapaths of dedicated hardware solutions sometimes added outside the processor to enhance application performance. By moving these datapaths into the processor, the system architect can discard the external control logic: the finite state machines and micro-sequencers. The processor and its software can provide this sequencing much more flexibly. Moreover, removing the function-specific control logic also eliminates most of the verification infrastructure necessary to test that logic and guarantees flexibility to accommodate new algorithms using the same datapath functions.

Moving the application-specific datapaths into the processor provides several other advantages. Fast instruction set extension and performance testing encourages, in turn, rapid prototype validation and experimentation. This allows the system designer to home in quickly on the best design for the target application set. The datapath is fully accessible from C/C++ code through the compiler using extended intrinsics and data-types. Storage

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Design Automation Conference. June 2001, Las Vegas, Nevada
Copyright 2001 ACM 1-58113-000-0/00/0000…$5.00.

elements (register files and special state registers) and pipeline flip-flops are generated by the TIE compiler in response to a high-level specification, and need not be created manually. Storage elements can be configured in width and number to adapt to the data precision and bandwidth requirements of the algorithm. The paradigm also simplifies the use of data memory, since the processor can simply share a unified data memory across many different tasks. This avoids the typical duplication of the assorted RAM structures, address generators, access ports and external interfaces found in designs that attempt to combine a range of specialized execution engines. Xtensa's RAM structures are configurable in type, depth and width beyond what the processor core requires, so as to support the width required by the added datapaths. So, this approach design time also reduces system cost through hardware sharing.

3. EXTENSION DESCRIPTION METHOD

To facilitate the new system design paradigm described in the previous section, Tensilica has introduced the Tensilica Instruction Extension (TIE) language for system designers to formally specify extensions to the Xtensa core processor. Like most previous machine description languages [4, 5, 6], TIE is an Instruction Set Architecture (ISA) description language. It relies on a tool, the TIE compiler, to generate an efficient hardware implementation and required additions to a suite of software tools, including the compiler, instruction-set simulator and debugger. TIE is not intended to be a complete processor description language. Instead, the TIE language provides designers simple ways to describe a broad variety of computational instructions, yet allows the TIE compiler to generate efficient hardware implementation for the instructions as an integral part of the processor. The language is simple enough for a wide range of designers to master, yet general enough to permit efficient description of complex instruction sets. As a result, the implementation efficiency matches that of a highly optimized processor core. In the rest of this section we briefly describe the capabilities of TIE language.

3.1 Instruction format and encoding

Instruction format and encoding can be specified through a combination of TIE *field*, *opcode*, *operand*, and *iclass* statements. A *field* statement simply gives a name to a group of bits in the instruction word. An *opcode* statement assigns instruction fields with values. An *operand* statement specifies how an instruction operand is encoded in an instruction field. An *iclass* statement defines the assembly format for an instruction. Since a TIE description defines extension instructions to the core Xtensa instruction set, there is a large set of pre-defined instructions fields, immediate fields and operands that can be used directly in the description. The following example completely defines the formats of two instructions, A4 and S4, which take two 32-bit input operands from the core register file, perform four 8-bit additions and subtractions, and store the result back to the core register file:

```
opcode A4 op2=0 CUST0
opcode S4 op2=1 CUST0
iclass RR{A4,S4}{out arr, in ars, in art}
```

The first two lines define the opcodes for A4 and S4 as sub-opcodes of a previously defined opcode CUST0 with the addition field op2 equal to 0 and 1 respectively. The second line makes use of the core-defined register operands arr, ars and art, and defines two new assembly instructions

```
A4 arr, ars, art
S4 arr, ars, art
```

3.2 Customized datapath

The computational part of an instruction is specified in a TIE *reference* block, which contains a series of assignment statements. The syntax of assignments is very similar to the Verilog hardware description language. The variables used in a reference block are either pre-defined, representing instruction operand values, or locally declared temporaries. For the A4 and S4 instructions defined in the previous section, their reference descriptions are

```
reference A4 {
    assign arr = {
        ars[31:24] + art[31:24],
        ars[23:16] + art[23:16],
        ars[15:8] + art[15:8],
        ars[7:0] + art[7:0]}
}
reference S4 {
    assign arr = {
        ars[31:24] - art[31:24],
        ars[23:16] - art[23:16],
        ars[15:8] - art[15:8],
        ars[7:0] - art[7:0]}
}
```

Even though the reference descriptions for the instructions are simple and direct, they may not yield the best hardware implementation, as in this example, where the logic for addition and subtraction should ideally be shared by the two instructions. To allow the specification of such logic sharing, TIE offers an alternative semantic statement that allows computations of multiple instructions to be specified in a single description block. For example, the semantics of A4 and S4 can also be described as:

```
semantic addsub {A4, S4} {
  assign arr = {
    ars[31:24]+(S4?~art[31:24],art[31:24])+S4,
    ars[23:16]+(S4?~art[23:16],art[23:16])+S4,
    ars[15:8]+(S4?~art[15:8],art[15:8])+S4,
    ars[7:0]+(S4?~art[7:0],art[7:0])+S4}
}
```

While the *semantic* statements allow more efficient hardware implementation, the *reference* statements are much simpler to write, more suitable for inclusion in documentation, and better for efficient simulation code generation. For these reasons, the TIE language allows the instruction semantics to be specified using either or both constructs. Typically, *reference* semantics are written first, and after the appropriateness of the instruction definition is verified, the implementation semantics are then coded.

3.3 Multi-cycle instructions

To keep up with the speed of the Xtensa core processor that is pipelined to run at high clock rate, instructions with complex computations require multiple clock cycles to complete. Writing and verifying multi-cycle implementations with efficient interlock

and data-forwarding logic is a challenging task for designers unfamiliar with the base processor's pipeline. TIE language provides a simple *schedule* construct to capture the multi-cycle requirement and relies on TIE compiler to derive the implementation automatically. For example, a MAC instruction that performs the following operation

```
acc = acc + a * b;
```

typically requires at least two cycles in an efficiently pipelined implementation. In order to achieve one MAC operation per cycle throughput, it needs to use `a` and `b` at the beginning of the first cycle, use `acc` at the beginning of the second cycle and produce a new `acc` at the end of the second cycle. This instruction timing can be easily specified in TIE as:

```
schedule MAC_SCHED {MAC} {
  use a 1; use b 1; use acc 2; def acc 2;
}
```

The rest of the implementation, including efficient insertion of pipeline registers, def-use interlock hardware, bypassing of results and generation of good code schedules are all handled by the TIE compiler.

3.4 Adding states and register files

Adding new state registers and register files can dramatically reduce the pressure on the core register file and increase the amount of data an instruction can send to and receive from the computational logic. For example, one could add state for the accumulator used by the MAC instruction. Without using the accumulator state, the MAC instruction would either take up three read ports and one write port of the register file in every clock cycle, or would have to take three cycles to finish the equivalent operations should the register file only have two read ports.

TIE states are extensions to the software-visible programming model. They allow instructions to have many more sources and destinations than are provided by the read and write ports of the core register files. They can be seen as analogous to the processor status registers or to states in finite state machines. They can also be used as dedicated registers holding values for some temporary variables during the program execution. Multiple temporary variables can be mapped to the same register if their live ranges are disjoint. When an application needs a large number of such sharable TIE states, it becomes more convenient to group the state registers into a register file and rely on the C compiler to assign variables to register entries. We discuss more about compiler support for TIE register files in section 4.

Describing instructions using a TIE state is straightforward. It involves declaring the state, specifying, in an instruction class, how the state is to be used, and describing the computational logic. The following TIE example completely describes a MAC instruction:

```
state acc 40 /* a 40-bit accumulator */
opcode MAC op2=0 CUST0
iclass RS{MAC}{in ars, in art}{inout acc}
reference MAC {
  assign acc = ars * art + acc;
}
```

Using register files involves one more step of describing register operands. For example we may wish to add a 24-bit register file for an application involving heavy processing of 24-bit graphics, using the following TIE statements:

```
regfile GR 24 16 /* 24-bit 16-entry */
operand gr r {GR[r]}
operand gs s {GR[s]}
operand gt t {GR[t]}
```

The three register operands use pre-defined instruction fields r, s and t as indices to access the register file contents. With this declaration, an instruction that averages two RGB pixel values can be simply described by the following:

```
iclass C {AVE} {out gr, in gs, in gt}
reference AVE {
  wire [8:0] r = gs[23:16]+gt[23:16];
  wire [8:0] g = gs[15:8]+gt[15:8];
  wire [8:0] b = gs[7:0]+gt[7:0];
  assign gr = {r[8:1],g[8:1],b[8:1]};
}
```

Note that the TIE language provides mechanism for creation of invisible state, that is, state not explicitly declared and directly accessible by the programmer. This dramatically reduces the opportunity for state-dependent errors and helps guarantee correctness and consistency of the processor in all of its implementations.

4. SOFTWARE SUPPORT

The Xtensa software system provides support for each configuration as seamlessly as software developed for traditional, non-configurable, systems. In many ways, it provides better support. In a traditional processor software developers are often forced to adapt their algorithms to constraints imposed by general-purpose programming languages targeted for general-purpose hardware. In contrast, configurable processor users can design their hardware and software development system together to better match the underlying algorithm. The key requirement for seamless support is completeness. When a user adds a custom TIE instruction, the Xtensa C/C++ compiler, assembler, simulator, debugger, operating systems and application libraries are all automatically modified to support the resulting architecture.

The Xtensa software system supports configurability through appropriate features in the TIE and the generation of dll's (dynamically linked libraries) and Xtensa target code from the TIE description. When a new instruction is added, TIE compiler generates dll's that describe the TIE instructions, typically in 30-60 seconds for large extensions. This speed is essential to support rapid instruction set architecture tuning and experimentation.

Every TIE instruction is directly accessible in C or C++ via an intrinsic function. In addition, the TIE language allows users to define new C datatypes that are mapped to TIE register files along with instruction sequences to load and store these datatypes from and to memory. The C/C++ programmer can use these types as if they were built-in data-types, declaring scalar variables, arrays or structures of them. Data operations are described via intrinsics, but register allocation, instruction sequences for loading and storing new datatypes, addressing arithmetic and control flow generation are all handled automatically just as native datatypes are. The generated dll's contain information about the side effects and pipelining of all TIE instructions, enabling the C/C++ compiler to schedule these instructions correctly and efficiently.

Programming with custom data-types is generally simpler than with pure C since algorithms directly map into appropriate operations.

The TIE compiler also generates operating system context switch code using the load and store instruction sequences for new datatypes, mentioned above, further automating software platform delivery. Commercial operating systems, such as Wind River's VxWorks™, are delivered pre-built with hooks in their context switching code to call the routines generated by the TIE compiler. The architecture and operating system code generator also supports *lazy context switching*. Register files and state can be grouped into coprocessor classes, each protected by an access flag. This way registers need to be saved and restored only when multiple processes actively share a particular coprocessor.

The same TIE semantic descriptions used to generate hardware are also used to generate the dll for the instruction set simulator. For each TIE instruction, a semantic function is generated containing callbacks into the simulator API for updating and evaluating simulated state. The simulator is able to correctly simulate an Xtensa program at close to one million instructions per second regardless of whether the simulated program uses TIE instructions or not. The TIE compiler also generates descriptions of custom register files so that the debugger is able to handle custom register files as cleanly as the base registers.

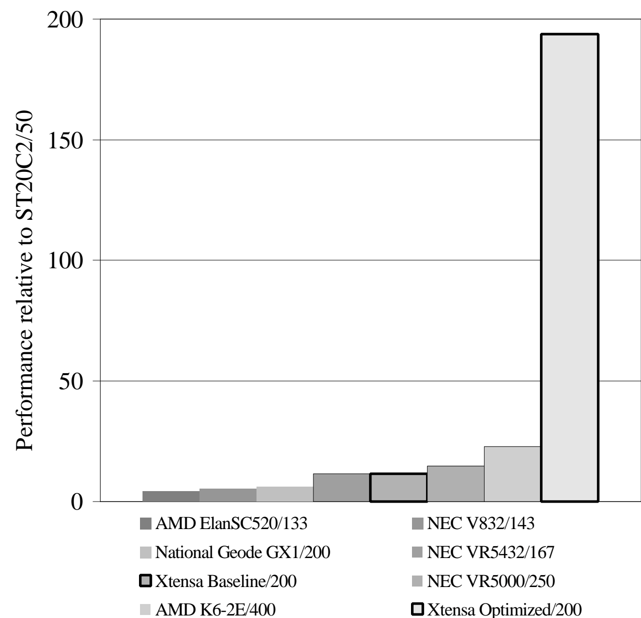
The compiler also supports vectorization, which allows ordinary scalar C code to fully exploit SIMD (single instruction, multiple data) or vector extensions, such as the TIE-based Vectra™ DSP engine, a vector DSP coprocessor. The compiler is able to automatically vectorize C code regardless of how many vector elements are configured into each Vectra register. Tensilica also provides hand-coded application libraries including FFTs, filters, convolution decoders, and other routines. Those routines are also automatically customized for each configuration of the Vectra engine.

5. APPLICATION EXAMPLES

Two sets of representative application kernels – one for consumer devices and one for telecommunications – help illustrate the impact of TIE-based configurability on application performance. The process of tuning these two suites of algorithms parallels the typical use of extensible processors – each of the algorithms is complex and a single processor is tuned for enhanced performance across the whole mix of tasks. Moreover, the twenty separate applications or test-cases in these two suites were ported, analyzed and used to drive processor configuration over a period of just eight weeks by one engineer, using Tensilica's standard tools.

5.1 Example 1: Consumer Multimedia

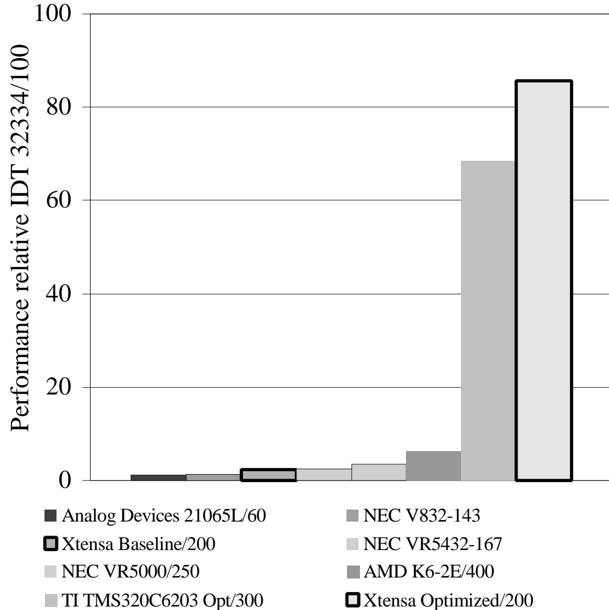
Video processing lies at the heart of consumer electronics – in digital video camera, in digital television and in games. Common tasks include color-space conversion, two-dimensional filters and image compression. The industry-standard EEMBC Consumer benchmark suite includes a representative sample of all these applications [7]. A baseline configuration of Tensilica's Xtensa processor already includes many appropriate features for these tasks, and even this baseline configuration at 200MHz delivers performance more than eleven times of a basic RISC processor, and on par with popular high-end 32-bit and 64-bit stand-alone processors, where performance is measured as the geometric mean of the relative number of iterations per second through each algorithm compared to the reference processor (ST Microelectronics ST20C2™ at 50MHz). However, when instructions for image filtering and color-space conversion (RGB-to-YIQ and RGB-to-CYMB) are added using TIE, the average performance is increased by a further 17 times, resulting in a processor with almost 200 times the performance of the reference processor as shown in Figure 1. The base configuration was optimized for 200MHz worst-case performance in 0.18µ CMOS technology and utilized 16KB two-way set associative caches, 256KB local data RAM, 16-entry write buffer and 32-bit multiplier for a total of 57,600 gates of logic. The optimized results used software that exploited the additional 64,100 gates of extensions implemented in TIE.



5.2 Example 2: DSP Telecommunications

Telecommunications applications present a different set of challenges. Here the data is often represented as 16-bit fixed-point values, as densely compacted bit-streams or as redundantly encoded channel data. Over the past ten years, standard DSP processors have evolved to address many of filtering, error correction and transform algorithms. The EEMBC "Telemark" benchmark includes many of these common tasks. In this case, the applications designer might start with a standard Xtensa configuration. This gives baseline performance on the EEMBC benchmarks that compares well with other leading 32-bit and 64-bit RISC processors, where performance is measured as the geometric mean of the relative number of iterations per second through each algorithm compared to the reference processor (IDT 32334™ – MIPS32 architecture - at 100MHz). However, when additional features are added, including the Vectra DSP co-processor and a few additional instructions in TIE, and the code is re-optimized to exploit the configuration, the performance jumps another 37 times. The overall performance then exceeds that a high-end Texas Instruments TMS320C6203™ VLIW DSP using

hand-optimized code, as shown in Figure 2. The base configuration was again optimized for 200MHz worst case performance in 0.18µ CMOS technology and used 16KB two-way set associative caches, 16-entry write buffer, but not the Vectra extensions. The optimized code utilizes Vectra and 18,000 additional gates of TIE for a total of 180,000 gates.



6. CONCLUSION

System-on-chip integration offers significant improvements in system bandwidth, cost and power efficiency, compared to systems build with discrete semiconductor building blocks. These application-specific system ICs can take full advantage of the efficiency benefits of application-specific processors, but designers can generate new processors quickly and completely to fit tight schedule requirements. Automatic generation of optimized processor core hardware and of software tools sharply reduces the time, cost and risk in development of new processor-based platforms, and eases the integration of processors into system-on-chip designs. New methodologies, tools, and processor foundations are required for this shift to processors. These methods enable significant new silicon platforms that combine the flexibility of standard programming models and the efficiency of application-tuned silicon.

Processor configurability has many dimensions — sizing of memories, addition of specialized external interfaces, and incorporation of closely coupled peripherals. The single most important dimension of configurability is instruction set extension. This paper has outlined the mechanisms for instruction set extensibility, the Tensilica Instruction Extension (TIE) description format and techniques for full configuration of compilers, simulators, RTOS and RTL implementation. The performance impact is significant, often averaging more than ten-fold that of current implementations of traditional, fixed-instruction set, general-purpose processors.

7. ACKNOWLEDGEMENT

The authors wish to thank the entire technical staff of Tensilica, whose efforts underlie the Xtensa Processor Generator, the foundation of this work. Special recognition is due to Michael Carchia, who did all the EEMBC benchmarking, and to Kim Alfaro and Pavlos Konas who reviewed the manuscript.

8. REFERENCES

[1] N. Zhang and R. W. Brodersen, "Architectural Evaluation of Flexible Digital Signal Processing for Wireless Receivers," Proc. Asilomar Conf., Pacific Grove, CA, October 2000

[2] R.G. Bushroe et al., "CHDS: A Foundation for Timing-Driven Physical Design into the 21st Century," SEMATECH, Inc.

[3] R. Gonzalez, "Configurable and Extensible Processors Change System Design". Hot Chips 11, 1999. ftp://www.hotchips.org/pub/hotc7to11cd/hc99/hc11_pdf/hc99.s4.3.Gonzalez.pdf

[4] G. Hadjiyiannis, S. Hanono, and S. Devadas. "ISDL: An instruction set description language for retargetability". Design Automation Conference, 1997

[5] V. Zivojnovic et al. "LISA - machine description language and generic machine model for HW/SW co-design". In IEEE Workshop on VLSI Signal Processing, 1996.

[6] A. Fauth, J. Van Praet, and M. Freericks. "Describing instructions set processors using nML". In Proc. European Design and Test Conf., pages 503-507, Paris (France), Mar. 1995.

[7] http://www.emmbc.org