Rapid SOC Development Using Automatically Generated Processors

October, 2007
The system architect faces a number of important decisions in creating the best SOC structure. Good choices early in the design process reduce silicon cost and power, increase system performance, and improve development and verification efficiency. This white paper explains the major decisions in architecting an SOC and guides the designer to a systematic approach to design structure using processors as the fundamental building blocks of the SOC. This design flow encourages wide use of processors as the default for implementing tasks and focuses on how to balance cost, performance, and flexibility within an SOC design framework. The foundations for the design flow are these:

1. Work top-down from the system’s essential I/O interfaces and computation requirements.

2. Use processors pervasively to implement tasks
   a. When tasks have specific computational patterns, optimize the processor to fit the tasks
   b. When a task exceeds the capacity of an optimized processor, parallelize the task across processors
   c. When a group of tasks fits together within a processor’s capacity limit, map the tasks together onto one processor to minimize hardware cost, power, and communications overhead.

3. Measure the communications traffic patterns and optimize the software and hardware interconnects around those patterns.

4. Start with early, rough simulation of communication tasks and refine the system into detailed implementations of processors, software, and other blocks, all running in increasingly accurate simulations.

**The Starting Point: Essential Interfaces and Computation**

The first system design step is identification of the chip’s essential input/output interfaces and computation. The target product’s marketing requirements usually establish the mandatory physical interfaces and necessary functions. These mandatory elements form the starting point for all other decisions—implementation decisions about these functions and decisions about inclusion of other supporting functions.

Not all interfaces and computations are equally fundamental to the design, of course. For example, in an integrated disk-drive controller, the external interface to the read-head and the servo motor are essential to the function, but external interface to buffer memory is not. Buffer memory could be implemented either on-chip or off-chip depending on more detailed analysis of cost and bandwidth tradeoffs. Similarly, a Secure Socket Layer security chip effectively requires implementation of the RSA (Rivest-Shamir-Adelman) algorithm for public/private key encryption, but may or may not include other TCP/IP protocol-processing functions.
Parallelizing a Task

When a single task shows particularly high computational demands, the designer must either use a faster general-purpose processor or pursue a more parallel hardware implementation. Fast general-purpose processors often require very high clock rates—for example, greater than 1 GHz. High clock-rate processors are likely to be both unacceptably power-hungry and difficult to design and integrate into an SOC without specialized skills. For embedded applications, an approach using parallel hardware resources is more likely to fit the embedded SOC’s needs.

Historically, hard-wired logic using RTL-based design was the only viable choice for parallel computation but this design approach typically limited the complexity of the algorithms that could be implemented. Extensible processors open up a simple but highly efficient means to exploit parallelism, especially for fine-grained (instruction-level) parallelism.

The basic algorithm analysis and instruction-design process is systematic but complex. Compiler-like tools that design application-specific instruction sets can automate much of this process. In fact, the more advanced compiler algorithms for code selection, software pipelining, register allocation, and long-instruction-word operation scheduling can also be applied to discovery and implementation of new instruction definitions and code generation using those instructions.

Automatic processor generation builds on the basic flow for application-specific processor generation but adds the creation of automatic processor architecture, as shown in Figure 1.

![Figure 1: Automatic Processor Generation Concept](image-url)
However, automatic processor generation offers benefits beyond simple discovery of improved architectures. Compared to human-designed instruction-extension methods, automation tools eliminate any need to manually incorporate new data types and intrinsic functions into the application source code and provide effective acceleration of applications that may be too large or complex for a human programmer to assess. As a result, this technology holds tremendous promise for transforming the development of processor architectures for embedded applications.

The essential goals for automatic processor generation are these:

1. A software developer of average experience should be able to easily use the tool and achieve consistently good results.

2. No source-code modification should be required to take advantage of generated instruction sets. (Note that some ways of expressing algorithms are better than others in exposing the latent parallelism, especially for SIMD optimization, so source code tuning can help. The automatic processor generator should highlight opportunities to the developer to improve the source code.)

3. The generated instruction sets should be sufficiently general-purpose and robust so that small changes to the application code do not degrade application performance.

4. The architecture design-automation environment should provide guidance so that advanced developers can further enhance automatically generated instruction-set extensions to achieve better performance.

5. The development tool must be sufficiently fast so that a large range of potential instruction-set extensions can be assessed—on the order of thousands of architectures per minute.

The requirement for generality and reprogrammability mandates two related use models for the system:

a) Initial SOC development: C/C++ in, instruction-set description out

b) Software development for an existing SOC: C/C++ and generated instruction-set description in, binary code out

Tensilica’s XPRES (Xtensa Processor Extension System) compiler implements automated processor instruction-set generation. A more detailed explanation of the XPRES flow will help explain the use and capability of this further level of processor automation. Figure 2 shows the four steps implemented by XPRES compiler. All of these steps are machine-automated, except for optional manual steps as noted.
The generation of a tailored C/C++ compiler adds significantly to the usefulness of the automatically generated processor. Even when the source application evolves, the generated compiler looks aggressively for opportunities to use the extended instruction set. In fact, this method can even be effective for generating fairly general-purpose architectures. So long as the basic set of operations is appropriate to another application, even if that application is unrelated to the first, the generated compiler will often use the extended architecture effectively.
The automatic processor generator internally enumerates the estimated hardware cost and application performance benefit of each of thousands of configurations, effectively building a pareto curve, such as that shown in Figure 3. Each point on the curve represents the best performance level achieved at each level of added gate count. This image is a screen capture from Tensilica’s Xplorer development environment, for XPRES results on a simple video motion-estimation routine (sum-of-absolute-differences).

**Figure 3: Automatic Generation of Architectures for Sum-of-Absolute-Differences**

Automatic generation of instruction-set extensions applies to a very wide range of potential problems. It yields the most dramatic benefits for data-intensive tasks where much of the processor-execution time is spent in a few hot spots and where SIMD, wide-instruction, and operation-fusion techniques can sharply reduce the number of instructions per loop iteration. Media- and signal-processing tasks often fall squarely in the sweet spot of automatic architecture generation. The automatic generator also handles applications where the developer has already identified key applications-specific functions, implemented those functions in TIE, and used those functions in the C source code. Figure 4 shows the results of automatic processor generation for three applications using the XPRES compiler, including one fairly large application: an MPEG4 video encoder.
<table>
<thead>
<tr>
<th>Application</th>
<th>Speed-up</th>
<th>Baseline Code Size</th>
<th>Code Size with Acceleration</th>
<th>MIPS32 Code Size (gcc –O2)</th>
<th>Configurations Evaluated</th>
<th>Generator Run Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG4 Encoder</td>
<td>3.0x</td>
<td>111KB</td>
<td>136KB</td>
<td>356KB</td>
<td>1,830,796</td>
<td>30</td>
</tr>
<tr>
<td>Radix-4 FFT</td>
<td>10.6x</td>
<td>1.5KB</td>
<td>3.6KB</td>
<td>4.4KB</td>
<td>175,796</td>
<td>3</td>
</tr>
<tr>
<td>GSM Encoder</td>
<td>3.9x</td>
<td>17KB</td>
<td>20KB</td>
<td>38KB</td>
<td>576,722</td>
<td>15</td>
</tr>
<tr>
<td>GSM Encoder (FFT ISA)</td>
<td>1.8x</td>
<td>17KB</td>
<td>19KB</td>
<td>38KB</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 4: Automatic Architecture Generation Results for DSP and Media Applications

The figure includes code-size results for the baseline Xtensa processor architecture and the automatically optimized Xtensa processor architecture for each application. Using aggressively optimized instruction sets generally increases code size slightly, but in all cases, the optimized code remains significantly smaller than that for conventional 32-bit RISC architectures. The figure also shows the number of configurations evaluated, which increases with the size of the application. The automatic processor generator run time also increases along with the size of the application, but averages about 50,000 evaluated configurations per minute on a 2GHz PC running Linux.

The figure also shows one example of generated-architecture generality. The GSM Encoder source code was compiled and run, not using an architecture optimized for the GSM Encoder, but for the architecture optimized for the FFT. While both are DSP-style applications, they have no source code in common. Nevertheless, the compiler automatically generated for the FFT-optimized processor could recognize ways to use the processor’s FFT-optimized instruction set to accelerate the GSM Encoder by 80% when compared to the performance of code compiled for the baseline Xtensa processor instruction set.

Completely automatic instruction-set extension carries two important caveats:

1. Programmers may know certain facts about the behavior of their application, which are not made explicit in the C or C++ code. For example, the programmer may know that a variable can only take on a certain range of values, or that two indirectly referenced data structures can never overlap. The absence of that information from the source code may inhibit automatic optimizations in the machine code and instruction extensions. Guidelines for using the automatic instruction-set generator should give useful hints on how to better incorporate that application-specific information into the source code. The human creator of instruction extensions may know this information and be able to exploit this additional information to create instruction sets and corresponding code modifications.

2. Expert architects and programmers can sometimes develop dramatically different and novel alternative algorithms for a task. A different inner-loop algorithm may lend itself much better to accelerated instructions than the original algorithm captured in
the C or C++ source code. Very probably, there will always be a class of problems where the expert human will out-perform the automatic generator, though the human will take longer (sometimes much longer) to develop an optimized architecture.

Summary

The implications of automatic instruction-set generation are wide-ranging. First, this technology opens up the creation of application-specific processors to a broad range of designers. It is not even necessary to have a basic understanding of instruction-set architectures. The basic skill to run a compiler is sufficient to take advantage of the mechanisms of automatic instruction-set extension.

Second, automatic instruction-set generation deals effectively with complex problems where the application’s performance bottleneck is spread across many loops or sections of code. An automated, compiler-based method is easily able to track the potential for sharing instructions among loops, the relative importance of different code sections based on dynamic execution profiles, and the cumulative hardware cost estimate. Global optimization is more difficult for the human designer to track.

Third, automatic instruction-set generation ensures that newly created instructions can be used by the application without source-code modification. The compiler-based tool knows exactly what combination of primitive C operations corresponds to each new instruction, so it is able to instantiate that new instruction wherever it benefits performance or code density. Moreover, once the instruction set is frozen and the SOC is built, the compiler retains knowledge of the correspondence between the C source code and the instructions. The compiler can utilize the same extended instructions even as the C source is changed.

Fourth, the automatic generator may make better instruction-set-extension decisions than human architects. The generator is not affected by the architect’s prejudice against creating new instructions (design inertia) or influenced by architectural folklore on rumored benefits of certain instructions. It has complete and quite accurate estimates of gate count and execution cycles and can perform comprehensive and systematic cost/benefit analysis. This combination of benefits therefore fulfills both of the central promises of application-specific processors: cheaper and more rapid development of optimized chips and easier reprogramming of that chip once it’s built to accommodate the evolving system requirements.