

# Trends and requirements in compiler technology for embedded cores

by Gerard Vink, Altium

THIS ARTICLE FOCUSES ON CODE-GENERATION TECHNIQUES FOR CONFIGURABLE CORES, EXAMINING THE ARCHITECTURE AND OPTIMIZATIONS APPLIED IN A COMPILER AS WELL AS THE FRAMEWORK TO BUILD SUCH A COMPILER. ALTIUM'S VIPER COMPILER FRAMEWORK IS USED AS A REFERENCE PLATFORM TO ILLUSTRATE CONCEPTS THAT ARE BEING DISCUSSED.

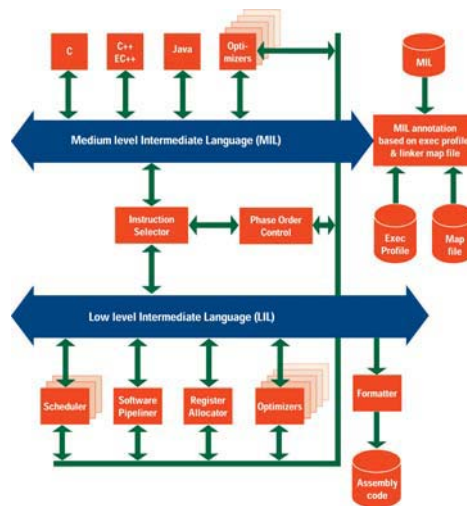


Figure 1. Altium's Viper compilers implement a basic architecture in which the compiler's front-end translates the C/C++ source code into a medium-level intermediate language where target-independent front-end optimizers operate. The instruction selector transfers MIL into a low-level intermediate language, where instruction scheduling, register allocation, and backend-specific optimizers operate.

New compilers support configurable architectures and can generate efficient instruction schedules for cores that support high levels of instruction and/or data parallelism. Today's leading compilers achieve a level of efficiency where it is no longer required to write performance-critical code in assembly instead of C. Although Altium's Viper compiler framework is designed to support the processor architectures of the 21st century, there seem to be remarkable advantages in execution speed and code size of about 10 and 40% when compilers for traditional 8 and 16-bit processors are built with Viper.

The performance of embedded processing cores is influenced by device architecture and device technology separately. Changes in device architecture have a major impact on compiler technology and occur roughly every ten years. The latest shifts in embedded device architecture were the introduction of VLIW and SIMD concepts to increase the levels of instruction and data parallelism. Continuous advancements in device technology enabled the construction of FPGA devices that are capable to store the circuitry of high performance processing cores. Furthermore, improvements in electronic design automation and the availability of off-the-shelf intellectual property building blocks facilitate the design of configurable application specific processing cores that can be implemented in either ASIC or FPGA.

Very-long-instruction-word (VLIW) cores offer high levels of instruction parallelism and may also employ data parallelism; typically impose restrictions on legal instruction-operand groupings that can be issued in parallel; and have no hardware facilities to optimize a given instruction schedule. A single-instruction-multiple-data (SIMD) core lets one instruction operate on multiple data items, which is especially productive for applications that process video images or audio files. Exploiting a core's features for concurrent processing surpasses the capabilities of previous generation embedded compilers.

Reconfigurable cores require frameworks (i.e., the environment, tools, and compiler specification languages) designed to facilitate the creation of compilers whose behavior can be specified at run-time to address changes and variants in the hardware. Another trend in embedded system architecture that requires dedicated compiler support is the introduction of cache and memory management systems that facilitate the implementation of large applications that execute under control of an operating system such as embedded Linux. Besides the technical challenges, a compiler designer has to deal with today's "economic realities". Therefore a framework for compiler construction should facilitate the construction of high-quality compilers in a timely and cost-efficient way.

The compiler framework packaged standard cores with a fixed architecture are not always the most optimal solution in terms of speed, power consumption, size, and economics. Therefore configurable cores exist which are typically customizable in terms of: instruction set and availability of hardware accelerators; number and mix of functional units; addressing modes and instruction/addressing-mode combinations; available number of registers; pipeline configuration as well as availability, size and "associativity" of data caches.

In the past, compilers for CISC and RISC architectures were successfully built by adjusting the target-specific parts of an existing compiler. However, today's variety in architectural features exceeds this approach, requiring new compiler frameworks to facilitate the creation of high-performance compilers. Such framework should be: modular – new components can be easily added and existing components can be removed, modified or replaced without impacting other components. The framework also should be tight – target-dependent source code should be limited to as few lines as possible, freeing the compiler writer to operate at a high level of abstraction. And the framework should be generic – target-dependent parts should be written almost entirely in purpose-built languages rather than C to maximize portability between architectures. The frame-

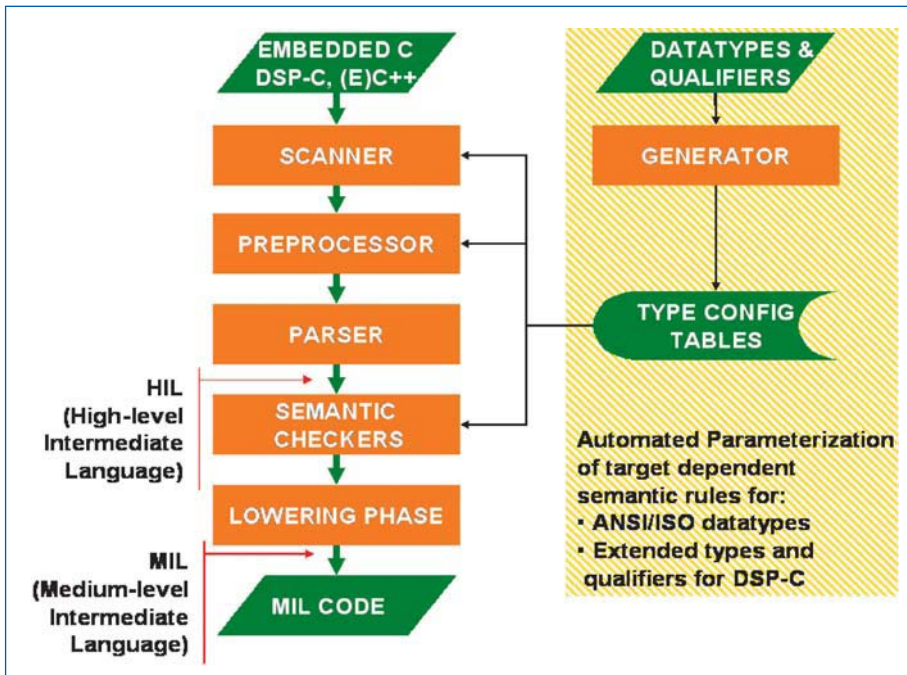


Figure 2. To build a target-specific front end, the compiler engineer describes the supported command line options, pragmas, datatypes and qualifiers using a dedicated language. The generator tools take this input and combine it with the target independent parts of the front end.

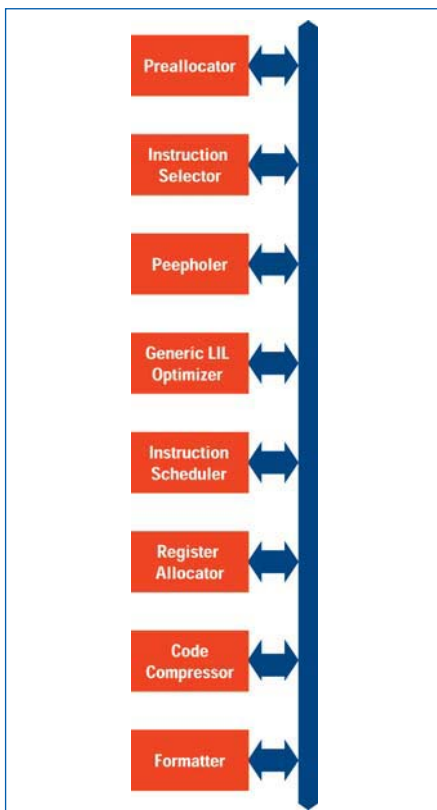


Figure 3. All back-end phases operate on the low-level intermediate language, which acts as a bus interface on which optimization phases can be plugged-in.

work should automatically check at compiler-creation time for errors in the specification. Within such a framework, a compiler with well-optimized code generators can be produced quickly, is easy to improve, maintain, and un-

derstand, and deals with processor variants in a natural and maintainable way. Reconfigurable cores require compilers that address changes and variants in the hardware at runtime.

The primary components of a compiler are the front- and back-end. A front-end reads the source files, performs lexical, syntactic, and semantic analysis, applies optimizations, and generates intermediate code. It depends primarily on the source language and is largely target-independent. The output of the front-end is a medium level intermediate language (MIL), which is a canonical representation of the source code that contains all the information needed for code generation, such as symbol tables and debug information. The back-end includes the compiler sections that depend on the target machine. First the instruction selector reads MIL input and translates it into a low-level intermediate language (LIL). The LIL objects are defined using a target description language (TDL) and correspond to a target processor instruction with the opcode, operands, and additional information used within the compiler. The back-end then continues with instruction scheduling, register allocation and optimizations that operate on the LIL. The output from the back-end is either assembly or object code. Figure 1 illustrates how the intermediate language representations MIL and LIL act as busses transferring information between optimization and code-generation phases. A set of local and global optimizers operates on MIL and LIL levels. The optimizers behavior and the order in which they are in-

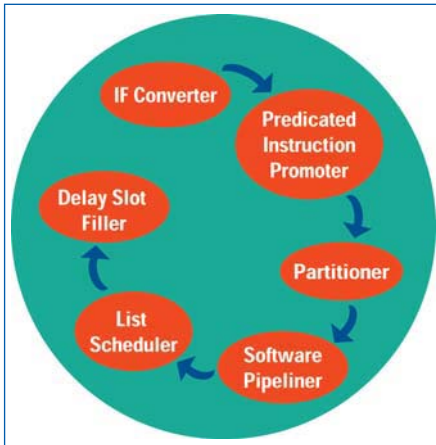


Figure 4. The instruction scheduler is a complex multiphase component that reorders the instruction sequence to optimally exploit the pipeline characteristics of a given target, which are described in the TDL language.

voiced are both target and application dependent and can be further influenced by data obtained from the linker map file and execution profiles.

The front-end ISO-C and C++ do not efficiently support architectural features such as multiple memory spaces, modulo or bit-reversed address modifications, and fixed-point arithmetic. To enable the compiler to employ these features, either C/C++ language extensions and/or a large set of intrinsic functions and pragmas are built into a compiler. Language extensions provide a suitable solution because they increase readability and maintainability of source code, allow the compiler to perform thorough static error checking, enable optimizers to exploit the parallelism of the target architecture, and simplify debugging. Language extensions enable the debugger to display fixed-point types in the correct format and to apply modulo and bit-reversed address modification when applicable.

A front-end applies various kinds of optimizations such as: classical optimizations, loop transformations, and processor- or application-specific optimizations. To efficiently translate control code, a large set of classical optimizations have to be applied. These classical optimizations each improve execution speed and/or code-size by a small percentage. Since more than twenty types of optimizations are typically applied, the effects on code quality can be significant.

Loop transformations are of particular importance in signal processing since many inner loops contain conditions that prevent the back-end from scheduling vector instructions (SIMD) or from executing instructions in parallel (MIMD). The loop optimizers rewrite the loop into a format more suited for parallel ex-

ecution, typically by applying transformations, such as loop-unrolling, reversal, interchange, fusion, fission, splitting, peeling, and switching. To make these transformations possible, data elements should be properly aligned, and supporting transformations such as induction variable rewriting, scalar expansion, and creation of temporary arrays, should be available as well.

The front-end can process multiple source modules simultaneously, and present the MIL for the entire application to the back-end, thereby widening the scope for applying global optimizations. Optimizations at the application wide level may include: global overlaying of constants, aggregation of global references, global inlining, alias analysis, value-range analysis, constant propagation, and code compression. Figure 2 shows how the Viper front-end is built for a given target architecture. The compiler engineer does not write any C code but writes two files that are passed to the front-end generator. The output of the generator is a set of C files that are linked together with the target independent parts of the Viper front-end. The primary components and stages that address a back-end designed to meet the complexities imposed by VLIW & SIMD architectures are illustrated in figure 3. Most components are generic and are either generated from or parameterized by target characteristics taken from a "target description file".

The instruction selector reads the MIL and symbol information created by the front-end and emits the corresponding LIL, the preallocator assigns a home for each automatic variable, reads and modifies symbols, and creates virtual registers. The peepholer traverses the LIL stream, making improvements as directed by a LIL editor pattern file. A generic LIL optimizer performs dead-code elimination, hoisting, and value tracking. Subsequently the instruction scheduler reorders the instructions. The register allocator chooses a physical register to use for each virtual register. The code compactor reduces code size by scanning for identical sequences of instructions and replaces all occurrences with a call to one single copy of the sequence. Finally the formatter reads the LIL-operations to generate assembly language output. An instruction scheduler (see figure 4) is a complex multiphase component that interacts with other parts of the back-end, such as the register allocator. When applicable for a given target the if converter replaces if-then-else constructs by predicated instructions. When allowed the predicated instruction promoter removes dependencies between LIL operations by removing the predicate from LIL operations. The partitioner separates targets with multiple register groupings. Next, the software pipeliner optimizes inner loop schedules, keeps track

of register pressure, and ensures it never exceeds the available resources. The list scheduler runs after software pipelining and again after register allocation to schedule any spill code. Finally, the delay slot filler addresses the various delay slot approaches of a processor.

Most back-end phases are used by every target, whereas other phases may be superfluous for a given architecture, for example instruction scheduling is not required for non-pipelined architectures. The requirements imposed by different core architectures on a backend phase may conflict to such extent that it is necessary to develop specific implementations for given classes of target architectures. Therefore the framework should enable the compiler engineer to modify or replace existing components without impacting other components. ■