TASKING®
UNVEILING THE
TASKING RISC-V COMPILER
RISC-V®
WHITEPAPER

UNVEILING THE TASKING RISC-V COMPILER

A Breakthrough regarding the development of FuSa and Cybersecurity compliant Software.

Author: Gerard Vink

INTRODUCTION

TASKING, a leading provider of innovative compiler solutions for the automotive functional safety sector, has introduced a compiler toolset for the RISC-V architecture. Amidst a crowded landscape of compilers, the question arises: could this offering significantly enhance the proliferation of RISC-V based System-on-Chips (SoCs) within the automotive industry? This whitepaper delves into this inquiry.

The first part of this paper highlights features of the RISC-V compiler and its use in FuSa related environments. The second part provides a comprehensive overview of TASKING's proprietary compiler technology framework, Viper. This framework serves as the backbone for all TASKING compilers and plays a pivotal role in the company's success within the automotive industry.

THE RISC-V COMPILER

The RISC-V instruction set is engineered to accommodate a diverse array of applications. With four ratified versions of the base instruction set architecture (ISA), each base ISA can be augmented with numerous ISA extensions. While certain ISA extensions, such as the "Standard Extension for Integer Multiplication and Division," notably influence compiler implementation, others, particularly those pertaining to memory protection, may initially appear to have no direct impact on a compiler but can introduce subtle side effects that invalidate certain compiler optimizations.

This versatility and configurability of RISC-V based System-on-Chips (SoCs) presents significant challenges for compiler developers. The compiler technology framework which underlies the implementation of a RISC-V compiler must exhibit a level of versatility and configurability commensurate with the ISAs and ISA extensions one intends to support. Compiler engineers, drawing from experience with various frameworks, widely regard Viper as the most adaptable compiler technology framework available.

In the process of re-targeting a Viper compiler, the conventional method of writing C/C++ code is bypassed. Instead, the majority of the compiler's components are crafted using domain-specific languages. This strategic approach streamlines the re-targeting process and minimizes errors. The translators responsible for converting domain-specific input into C/C++ meticulously scrutinize the input specifications for completeness and accuracy, ensuring early versions of the compiler excel in code correctness and suitability for FuSa-related development. The final compiler executable is generated through a C/C++ to machine code compilation, enabling the application of various optimization techniques to enhance compiler execution speed.

Utilizing the compiler toolset in a FuSa-related context presents specific demands on the toolset itself, its documentation, and the supporting organization. Drawing from over 30 years of experience in the automotive sector, TASKING is well-versed in meeting these rigorous requirements. The figure below illustrates the array of FuSa-related products offered that support your organization in complying with FuSa and Cybersecurity regulations and standards over the whole lifecycle of your products.

Figure 1, TASKING Safety Lifecycle Support

TASKING takes great pride in the maturity of their FuSa and Cybersecurity documentation, which accompanies the compiler. Notably, TASKING compilers were the pioneering recipients of certification against the ISO/SAE 21434 standard. This commitment extends beyond the compiler tool itself; also, the associated libraries are undergoing certification processes against ISO 26262 and IEC 61508. The maturity and structured format of the FuSa and CySec information empower our customers to fast-track their tool qualification processes, achieving compliance with FuSa and Cybersecurity standards with minimal effort and cost.

THE VIPER COMPILER TECHNOLOGY FRAMEWORK

The Viper Compiler Technology Framework has been developed to facilitate cost efficient construction of highly optimized compilers for Very Long Instruction Word (VLIW) architectures offering instruction-level and data-level parallelism. The features of modern RISC and CISC ISAs with optional DSP extensions are a subset from the VLIW domain and are as such supported by the Viper framework. Note that the following text describes the compiler technology framework, therefore the text is not RISC-V specific.

Efficient machine code can only be generated by a compiler that is precisely tailored to the configuration -- the ISA, ISA extensions, and instruction pipeline behavior -- of the target processor. In the context of reconfigurable IP, only a compiler that is retargetable, i.e. configurable for different target systems at compile-time (versus compiler build time) can exploit modifications made to the processor core. In addition to the complexity introduced by hardware configurability, compiler development is further complicated by facilities such as instruction-level parallelism (ILP), single-instruction-multiple data (SIMD), and non-orthogonal instruction sets.

TASKING's Viper compiler framework provides a versatile environment for the fast and efficient development of compilers which can be easily adapted to specific core architecture variants. As *Figure 2* shows, TASKING compilers are divided into two main components: The front-end and the back-end (also known as the code generator). The front-end performs lexical and syntactic analysis, semantic analysis, optimization, and generation of the intermediate code. A characteristic of the front-end is that its behavior is mainly determined by the source language and is largely independent of the target system. The intermediate code created by the Viper front-end is referred to as medium level Intermediate Language (MIL). MIL is a canonical equivalent of the source code and contains all the information required for code generation (e.g., symbol tables and debug information).

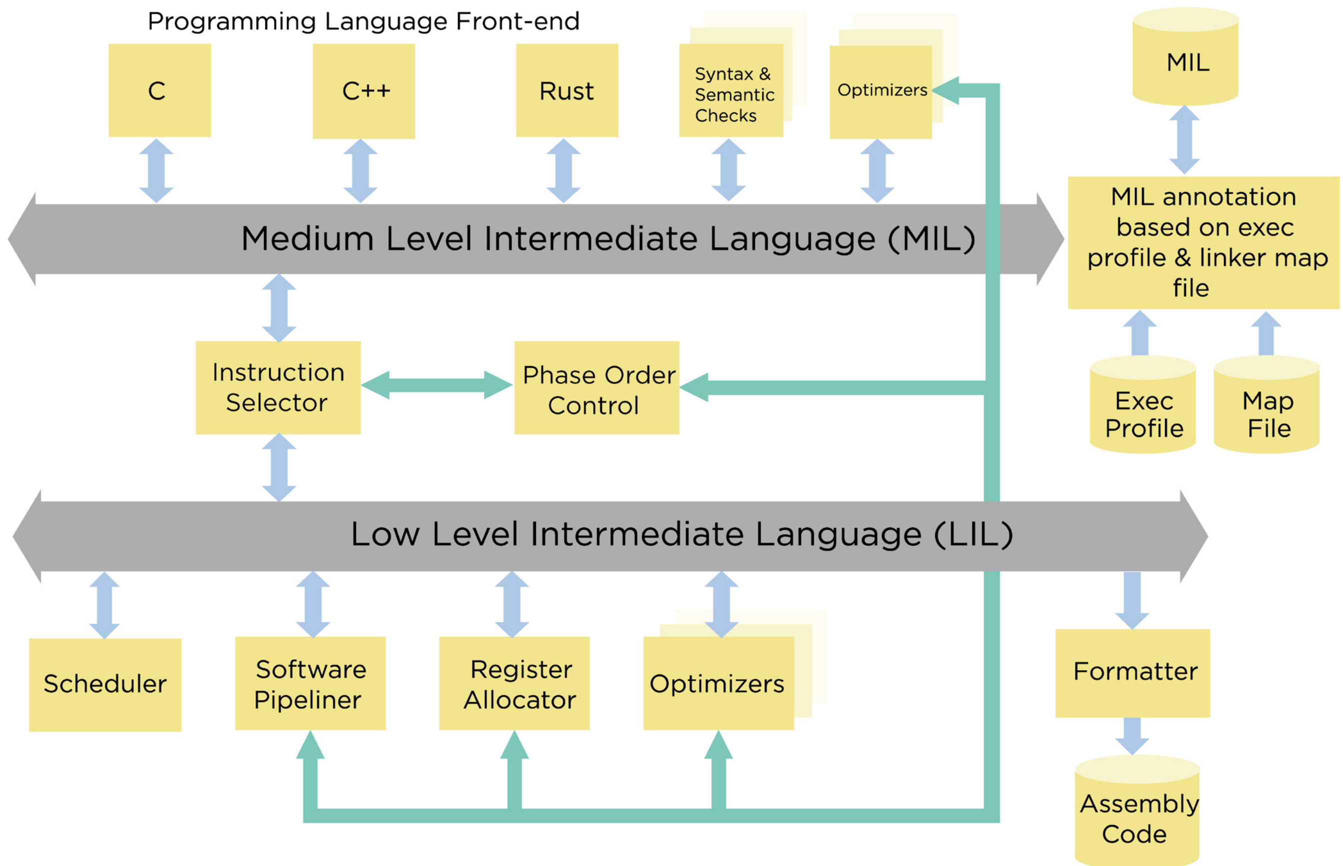

Figure 2, The Viper Compiler Technology Framework

The back-end contains those sections of the compiler which must be aligned to the target system. The first back-end phase, the Instruction Selector, converts the MIL code from the front-end to the Low-level Intermediate Language (LIL). The LIL objects (Lilops) are defined using a domain specific language called Target Description Language (TDL). Lilops refer to an instruction of the target processor with opcode, operands, and information used by the compiler such as instruction throughput, instruction latency, and pipeline dependencies.

The next steps of the back-end are instruction scheduling, register allocation and optimization at the Low Intermediate Language level. The final back-end phase is the Formatter which produces assembly code. The intermediate languages MIL and LIL act as buses that transfer information between the optimization and code generation phases.

The Viper compiler construction framework offers a large amount of local and global optimizers that operate at the MIL or LIL level. The behavior of the optimizers and the order in which they are called, managed by the Phase Order Control component, depend on the target system and the application software. The compilation process can be further steered by feeding the compiler with data taken from the linker map file and the execution profiles.

THE VIPER FRONT-END

C and C++ front-ends and a prototype Rust front-end are available.
The following features are provided by the front-end:

- DSP C/C++ language extensions,
- User-specifiable calling conventions,
- In-line assembly code,
- Intrinsic functions,
- Conventional optimizations,
- Application-wide optimizations,
- Loop transformations,
- Feedback loops to control register pressure
- Processor and application-specific optimizations.

BACK-END (CODE GENERATION)

The optimal utilization of instruction-level and data-level parallelism is anything but trivial even with orthogonal, homogeneous RISC architectures. With DSPs, however, code generation proves to be significantly more complex and requires other techniques. Register allocation is often complicated by the heterogeneity of the register set. Instruction scheduling and software pipelining are no longer separate stages but must interact with instruction selection and register allocation. The Viper back-end is designed to handle the DSP-specific complexities described above.

The main components and sections of the back-end *(Figure 3)* are:

- The Preallocator allocates storage for local variables, reads and modifies symbols, and sets up virtual registers.
- The Instruction Selector reads the MIL and symbol information generated by the front-end and generates the corresponding LIL.
- The Peepholer uses the LIL Editor to traverse the LIL stream and makes improvements according to a LIL Editor pattern file.
- The Generic LIL Optimizer performs many different optimizations, due to the generic specification of Lilops in TDL, the implementation of optimization algorithms does not need to be customized for a specific target architecture.
- The Instruction Scheduler is a group of code generator stages that organize the Lilops into an order optimal for the target processor. The Instruction Scheduler usually executes before and after register allocation.
- The Register Allocator assigns a physical register to each virtual register. Feedback loops between the back-end and front-end prevent front-end optimizations from causing excessive register pressure, which is detrimental to the overall result of optimizations.
- The Code Compressor reduces the code size by searching for identical instruction sequences. If a sequence occurs sufficiently often, all instances of this sequence are replaced by a call to a single existing copy.
- The Formatter reads the LIL operations to generate the assembler code. It forms the last stage of the code generator.

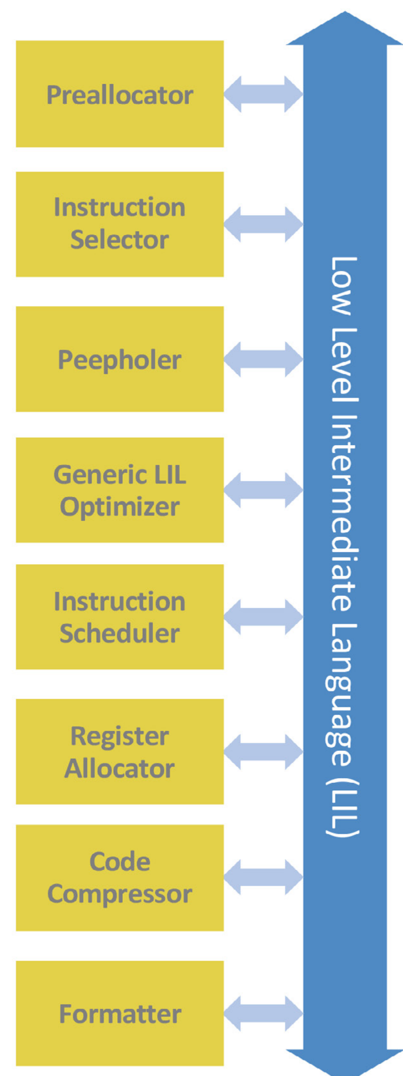

Figure 3, The main components and sections of the Viper back-end

The Instruction Scheduler is a complex, multi-phase component that interacts with other sections of the back-end (e.g., the Register Allocator) to generate optimal instruction sequences for a specific target system.
The scheduler can consist of the following components:

- The IF Converter replaces If-Then-Else constructs with Predicated Instructions, which can be processed more efficiently.
- The Predicated Instruction Promoter removes dependencies between LIL operations.
- The Partitioner is useful for target systems with two or more register sets, each of which is assigned to one or more functional units.
- The Software Pipeliner optimizes the schedules of the inner loops, tracking the register usage for each register class to avoid causing register spilling. It is an important component to facilitate auto-vectorization.
- The List Scheduler implementation is independent of the target system. All target system-specific information is taken from the TDL file of the target system. The List Scheduler runs after the software pipelining and again after the register allocation so that any spill code can be scheduled.
- The Delay Slot Filler executes after register allocation, and supports the various delay slot concepts for those processors where this is beneficial.

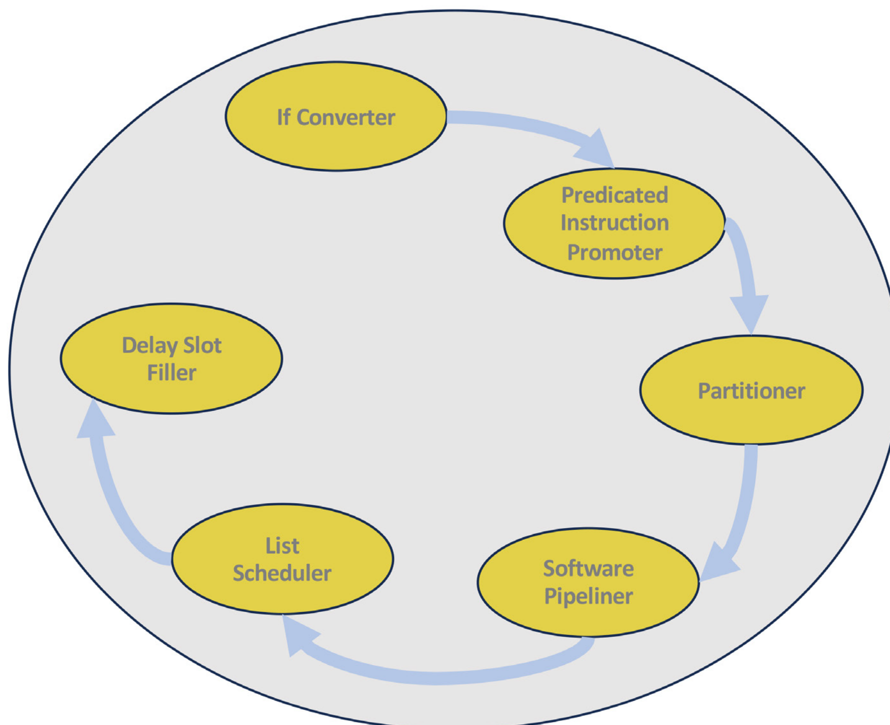

Figure 4, The main components of the Instruction Scheduler

LANGUAGE EXTENSIONS AND OPTIMIZATIONS

Although the ISO-C and C++ programming languages are designed for developing applications that interact with the underlying hardware, there is a lack of support for typical DSP architecture features. These features include, for example, complex memory systems, special addressing modes, fixed-point arithmetic and SIMD facilities. DSP-C was developed with the aim of meeting the special requirements of digital signal processing at the C level. It is an extension of the ISO/IEC specification 9899:2011(E) for the C language.

With the goal of meeting the code density and performance requirements of DSPs, Viper supports DSP-C language extensions, classical optimizations, application-wide optimizations, target system-dependent optimizations and loop transformations. Massive gains in execution speed can be achieved by exploiting the parallel data and instruction processing supported by DSPs -- this is one of the key features of the Viper technology --.

CONFIGURABLE IP CORES AND SOCS

The Viper Framework allows building compilers whose behavior can be specified at compile-time to allow changes and adjustments on the hardware side. Major architecture modifications such as the introduction of new instructions and addressing types or the introduction of a new pipeline architecture require the creation of a new compiler.
The following changes can be made directly without creating a new compiler:

- Removal of opcodes, registers and addressing types,
- Changing the quantity and composition of the functional units.

PROCESS MATURITY, FUNCTIONAL SAFETY, AND CYBERSECURITY

Product development at TASKING is carried out using Automotive SPICE level 2 compliant processes, which ensure predictable and repeatable product quality. The creation of evidence required for functional safety and cybersecurity certification of the compiler and associated libraries is seamlessly integrated into the development process. Consequently, the first version of a compiler is a high-quality product suitable for the development of safety-critical and security-related software.