VIPER

COMPILER TECHNOLOGY FROM ALTIUM

TASKING™
Embedded software development from Altium™

HIGHLIGHTS

- **Configurable IP cores and SoC support**
- **Modular retargetable design**
- **Description-based approach for retargeting and optimizations**
- **SIMD and MIMD optimization techniques**
- **VLIW-specific instruction scheduling**
- **Application-wide global optimizations**
- **Map file and execution profile controlled optimizations**
- **DSP-C specific language extensions**
- **ISO/IEC 9899:1999(E) C language**

RETARGETABLE COMPILERS ESSENTIAL FOR MICROPROCESSOR-BASED AND SoPC APPLICATIONS

The continuous growth in the use of electronics in products creates an increasing demand for different types of specialized and complex microprocessors that cater for specialized areas of design, such as automotive and telecommunications. In recent years the complexity and range of available microprocessors supplied by semiconductor manufacturers has grown exponentially to meet this demand.

Even with the large number of different and specialized microprocessors now available, their standardized nature means that often electronics engineers are unable to find a microprocessor that has the exact optimizations for speed, power consumption, size, economics, and instruction set that they are looking for.

Semiconductor manufacturers have responded to this need in two ways. First, they have fuelled the proliferation of new and specialized architectures. Secondly, they have made virtual IP cores for many of their microprocessors available to companies for use in SoC or SoPC designs, giving engineers and embedded software developers the ability to modify and customize microprocessor cores specifically for their design. This is particularly prevalent with specialized DSP cores.

At the same time, electronics designers and developers must create increasingly complex applications under shortening time-to-market pressures. This drives the demand for highly efficient compilers that can generate high code density, ensure code reliability, and maximize application speed.

Compilation of efficient machine code cannot be accomplished without a compiler that is optimized for the specific configuration of the microprocessor's architecture. In the context of reconfigurable IP, unless the compiler is retargetable, it will not be able to take advantage of the customizations that have been made. Code generation is further complicated by instruction and data level parallelism, heterogenous architectures, non-orthogonal instruction sets, hardware MAC units and zero-overhead loop facilities. To date, the development tool industry has struggled with the twin challenges of maintaining code compilation efficiency while providing full coverage across the wide spectrum of processor derivatives.

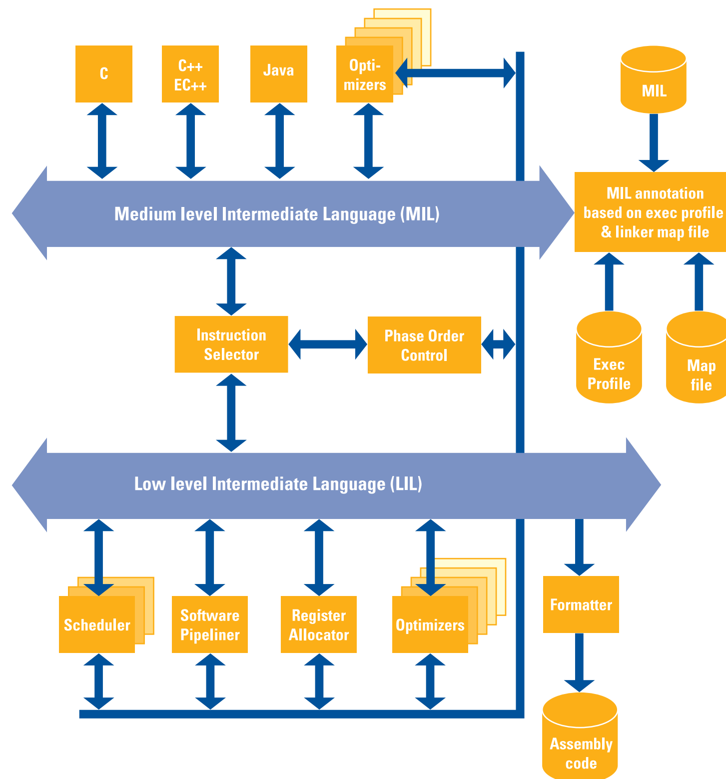
The TASKING Viper compilation environment addresses these challenges by providing a versatile framework for the efficient and fast development of compilers that can be easily retargeted to specific architecture variations.

The goals of the Viper compiler technology framework are to:

- **Create highly efficient compilers which:**
  - **generate the highest code density**
  - **ensure code reliability**
  - **maximize execution speed**
- **Accelerate retargeting to new architectures**
- **Facilitate the creation of compilers for configurable IP cores**

COMPILER

- C, C++ support
- DSP-C/C++ language extensions
- Vectorization techniques
- VLIW specific instruction scheduling
- Application-wide global optimizations
- User-specifiable calling conventions
- Inline assembly code
- Extensive set of pragmas and memory-type qualifiers
- Predicated execution
- Aggressive software pipelining
- Instruction mutation
- Delay slot filling
- Customer proprietary optimizations plug-in facility



Compiler front-end and back-end

HIGHLY EFFICIENT COMPILERS

The primary components of the Viper compiler are the front-end and back-end (also known as the code generator). The **front-end** performs lexical and syntactic analysis, semantic analysis and the generation of intermediate code. The front-end depends primarily on the source language and is largely target-independent. The intermediate code emitted by the Viper front-end is called **MIL** (Medium level Intermediate Language). The MIL is a canonical representation of the source code and contains all the information needed for code generation, such as symbol tables and debug information.

The **back-end** includes the compiler sections which depend on the target machine. During the first stage of processing, the instruction selector reads MIL input and translates it into Low level Intermediate Language (LIL). The **LIL** objects ("lilops") are defined using **TDL** (Target Description Language), and correspond to a target processor instruction, with an opcode, operands, and information used within the compiler.

The back-end then continues with instruction scheduling, register allocation and optimizations which operate on the LIL. The output from the back-end is assembly source code.

The inter-mediate language representations, MIL and LIL, act as buses that transfer information between optimization and code generation phases.

Viper supports an advanced set of local and global optimizers that operate on either MIL or LIL level. The behavior of the optimizers and the order in which they are invoked is both target- and application-dependent. The optimization process can be further influenced by providing the compiler with data obtained from the linker map file and execution profiles.

Front-end

Front-ends are available for C, C++ and EC++ that support TASKING DSP-C language extensions.

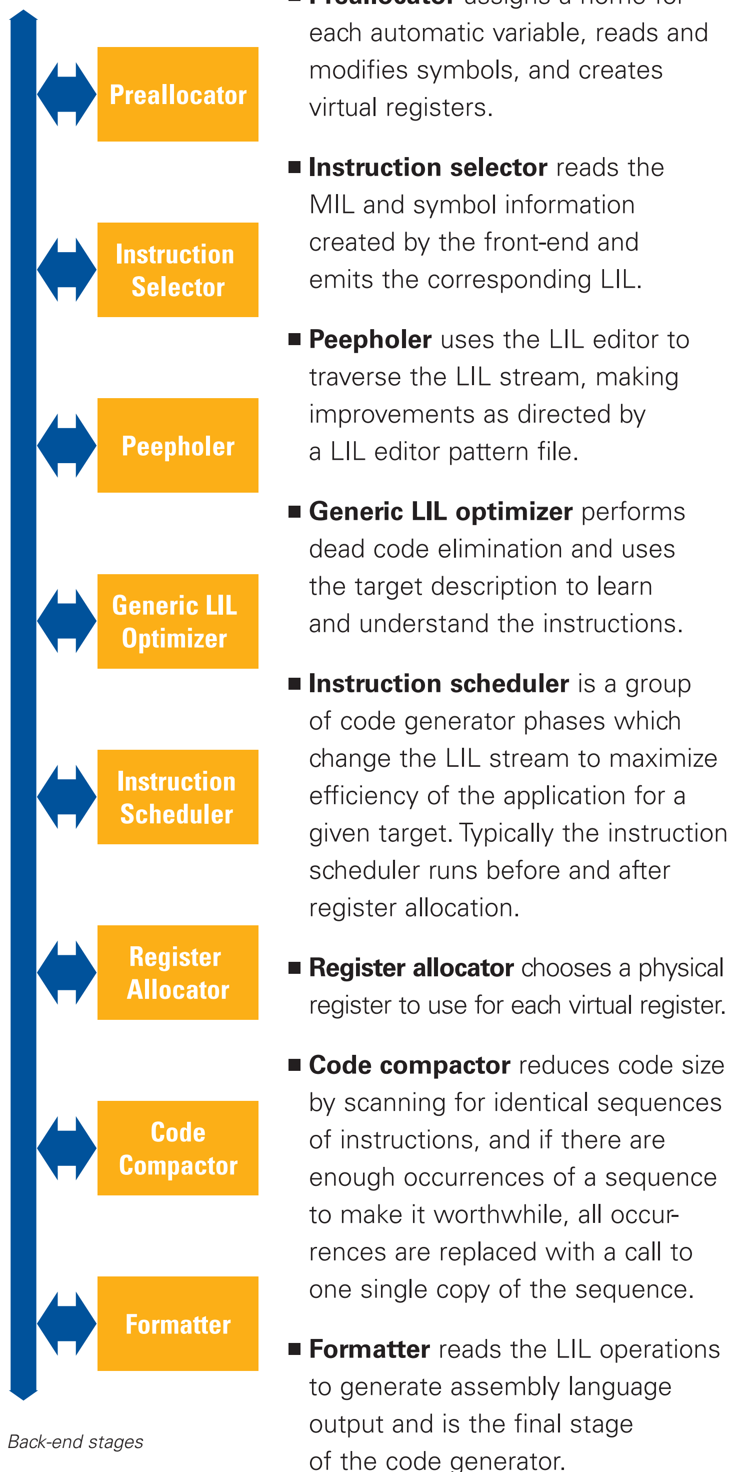
The front-end supports the following capabilities:

- **DSP C/C++ language extensions**
- **User-specifiable calling conventions**
- **Fully optimized inline assembly code**
- **Intrinsic functions**
- **Classical and application-wide optimizations**
- **Loop transformations**
- **Feedback loops**
- **Processor or application-specific optimizations**

Back-end

Optimal exploitation of instruction level parallelism and delay slot filling is not trivial for orthogonal homogeneous RISC architectures, but code generation for DSPs is far more complex and requires different techniques. Register allocation is complicated by the heterogeneous nature of the register set. Instruction scheduling and software pipelining are not separate phases anymore, but must interact with instruction selection and register allocation.

The Viper back-end is designed to address the DSP complexities previously described. The following are the primary components and stages of the back-end:

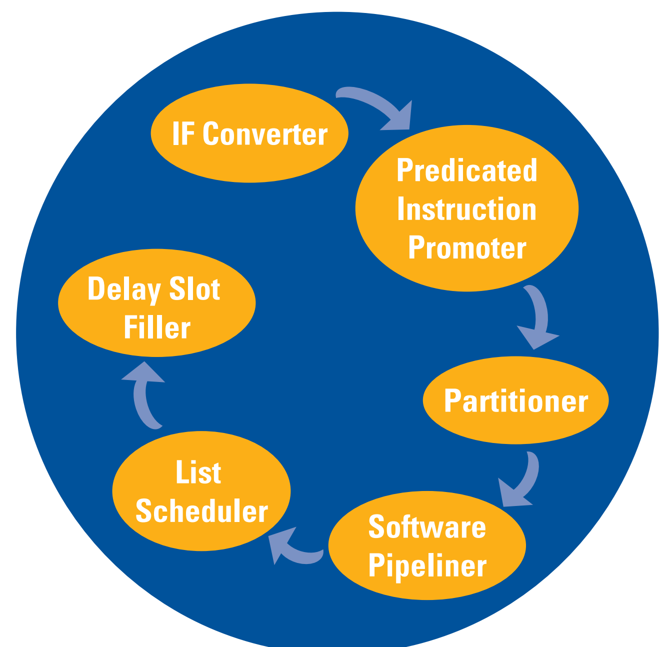


Back-end stages

- **Preallocator** assigns a home for each automatic variable, reads and modifies symbols, and creates virtual registers.
- **Instruction selector** reads the MIL and symbol information created by the front-end and emits the corresponding LIL.
- **Peepholer** uses the LIL editor to traverse the LIL stream, making improvements as directed by a LIL editor pattern file.
- **Generic LIL optimizer** performs dead code elimination and uses the target description to learn and understand the instructions.
- **Instruction scheduler** is a group of code generator phases which change the LIL stream to maximize efficiency of the application for a given target. Typically the instruction scheduler runs before and after register allocation.
- **Register allocator** chooses a physical register to use for each virtual register.
- **Code compactor** reduces code size by scanning for identical sequences of instructions, and if there are enough occurrences of a sequence to make it worthwhile, all occurrences are replaced with a call to one single copy of the sequence.
- **Formatter** reads the LIL operations to generate assembly language output and is the final stage of the code generator.

The instruction scheduler is a complex multi-phase component that interacts with other parts of the back-end, such as the register allocator, to create optional instruction schedules for a specific target. The scheduler can consist of the following components:

- **IF converter** replaces if-then-else constructs by predicated instructions which are more effectively handled by the pipeline.
- **Predicated instruction promoter** removes dependencies between LIL operations by removing the predicate from LIL operations.
- **Partitioner** is useful for targets with two or more register groupings, each associated with one or more functional units.
- **Software pipeliner** optimizes inner loop schedules, keeps track of register pressure for each class of register, and ensures it never exceeds the available resources (to prevent spilling). The software pipeliner is based upon Rau's Iterative Modulo Scheduling algorithm in combination with Modulo Variable Expansion.
- **List scheduler** is target-independent, and all target information is received from the target's TDL file. It runs after software pipelining and again after register allocation in order to schedule any spill code.
- **Delay slot filler** runs after register allocation and addresses the various delay slot approaches of a processor.



Instruction scheduler

LOOP OPTIMIZATIONS

- **Data alignment**
- **Induction variable rewriting**
- **Node splitting**
- **Live range splitting**
- **Scalar expansion, use of temporary arrays**
- **Loop unrolling, reversal, interchange, fusion, fission, splitting, peeling, jamming, switching, unwinding**

LANGUAGE EXTENSIONS AND OPTIMIZATIONS

Although the ISO-C and C++ programming languages are designed to support development of applications which interact with the underlying hardware, support for typical DSP architectural features is not present. These features include complex memory systems, address generation units, fixed-point arithmetic, and data level parallelism.

To meet the code density and performance needs of DSPs, Viper supports proprietary DSP-C extensions, classical optimizations, application wide optimizations, target-dependent optimizations, and loop transformations. However, exploiting a DSP's data or instruction level parallelism can improve execution speed by 100% to 500%, so this is a primary area of focus for Viper.

TASKING DSP-C evolved to meet the specific needs of DSP at the C language level. It is an extension to the ISO-C standard: ISO/IEC 9899:1999(E), Programming languages-C specification. TASKING developed proprietary DSP-C extensions to support DSP characteristics and issues not currently addressed by this standard.

```
#define FIR_ORDER 20
__fract __ds0 __circ         samples[FIR_ORDER ];
__fract __ds1                coeffs[FIR_ORDER ];

/*                           N
 * Computes result =        SUM(coef[i] * sample[T-i])
 *                          i=0
 */
__lfract fir_filter(int n)
{
          __fract __ds0 __circ *buffer_p =samples;
          __lfract             result = 0;
          int i;

          buffer_p += n;
          for(i=0; i<FIR_ORDER; i++)
          {
                  result += *buffer_p— * coeffs [i];
          }
          return(_round(result));
}
```

This example illustrates a FIR filter using C-language extensions

The DSP-C extensions:

- **Enable the use of fixed-point data types, circular and bit-reversed addressing and memory-type qualifiers at the C/C++ language level**
- **Provide application portability**
- **Allow bit-accurate simulation on a host with minimal impact on the DSP-C code**
- **Reduce the need for the use of intrinsic functions**
- **Increase readability and maintainability of source code**
- **Allow the compiler to perform thorough static error checking**
- **Enable optimizers to fully exploit the target architecture parallelism (memory system, scheduling instructions)**
- **Simplify debugging by enabling the debugger to display fixed-point types in correct format, and modulo and bit-reversed address modification are automatically applied**

Classical optimizations

A large set of classical optimizations is necessary for control code because each optimization typically improves execution speed and/or code-size by a small percentage.

The following classical optimizations are supported by Viper:

- **Expression simplification**
- **Constant folding**
- **Dead code elimination**
- **Jump chaining**
- **Switch simplification**
- **Conditional jump reversal**
- **Code reordering**
- **Constant propagation**
- **Dead store elimination**
- **Inlining**
- **Partial-redundancy elimination (includes CSE)**
- **Tail merging**
- **Copy propagation**
- **Code hoisting**
- **Localizing global variables**
- **Strength reduction**
- **Alias disambiguation**
- **And more ...**

Loop transformations

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 that is more suited for parallel execution. To fully exploit parallelism, Viper supports the listed loop optimizations.

Application-wide optimizations

Viper has the ability to run the front-end over each input module, and then present the MIL for the entire application to the MIL optimizers and the back-end.

The following are the type of optimizations applied at an application-wide scope:

- **Global overlaying of constants**
- **Aggregation of global references**
- **Global inlining**
- **Inter-procedural alias analysis**
- **Inter-procedural constant propagation**
- **Procedure sorting**
- **Remove unused functions**
- **Code compression**

ACCELERATE RETARGETING TO NEW ARCHITECTURES

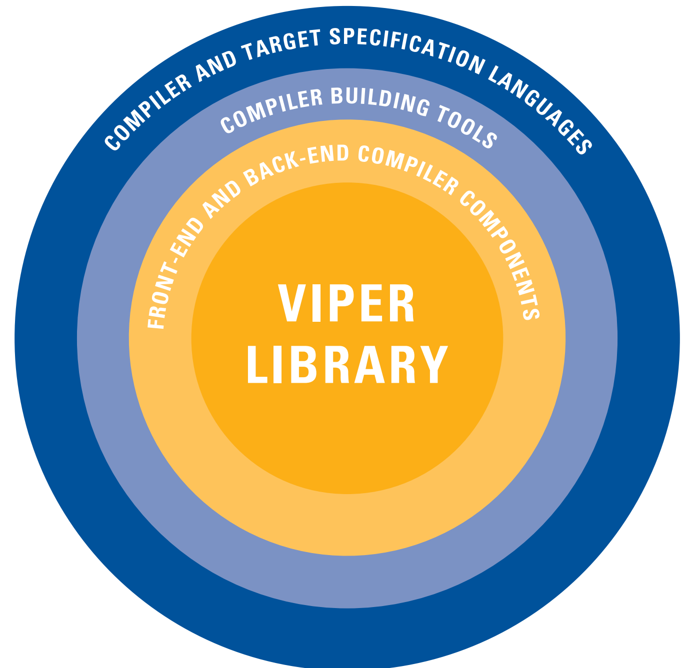
The Viper compilation environment is an extensible framework for developing quality high-performance compilers. The two inner layers constitute the actual compiler (the front-end and back-end). The two outer layers represent the tools and languages used to build the compiler.

Layer 1 is the Viper Library, which is a collection of target-independent functions that facilitate creating compiler phases, such as functions for creating and accessing register interference graphs and data dependency graphs.

Layer 2 consists of the front-end and back-end compiler components.

Layer 3 contains the tools to build a compiler.

Layer 4 contains compiler and target-specific languages. These are specially designed languages to specify target characteristics, the language implementation and compiler configuration. The Target Description Language (TDL) is one of the languages used in this layer. It is a flexible, object-oriented language for describing a target processor and is also used by the back-end to describe the incoming intermediate language.



Viper Compilation Environment

Configurable IP core and SoC support

The Viper framework is also designed to facilitate the creation of compilers whose behavior can be specified at run-time to address changes and variants in the hardware. Major architectural changes, such as the introduction of new instructions, new addressing modes, or a new pipeline architecture require the building of a new compiler. However, deletion of opcodes, registers and addressing modes, and changes in number and mix of functional-units can be made readily without rebuilding the compiler.

```
section Register {
        default_values {
                dwarf_code = 0;
                hard_value = 0;
                is_hard = 0;
                is_saved_by_callee = 0;
                kids = [];
        }

       columns = dwarf_code, width, allocatable;
       : a0       { 4;        40;     1; kids=[a0l,a0h,gb0]; }
       : a0l      { 5;        16;     1; }
       : a0h      { 6;        16;     1; }
       : gb0      { 0;        8;      0; }
       : r0       { 32;       16;     1; resources = [G1R G1W AU_UNIT0]; }
```

TDL example of target register specification

TASKING™
Embedded software development from Altium™

The overall modular design of Viper meets the goals of extensibility accelerated retargetting, and facilitated support for IP cores and SoCs.

Viper's module design provides the ability to:

- **Add new components for capabilities such as new optimization techniques or position-independent code for a target**
- **Modify or replace existing components without impacting other components**
- **Not utilize a component for a specific architecture**

In addition to its compiler technology, TASKING also provides the necessary assembler, linker and debugger technology that fully supports and exploits the features offered by Viper.

TASKING LEADERSHIP

Altium's TASKING brand carries over 25 years worth of experience in the embedded systems industry and is used by the world's leading telecom, datacom, wireless, automotive, avionics, aerospace, and computer peripheral manufacturers. This pedigree puts Altium in a unique position to play a lead role during the next wave of growth in embedded applications.

Through its TASKING brand, Altium takes a leadership position in the embedded software development industry by defining and developing language extensions, user interfaces and new state-of-the-art technologies to meet the demands of today's development challenges. The evolution of TASKING compiler, assembler, linker/locator, and debugger components fully consider the diverse needs of customers who rely on maximum performance from selected processor, DSP, IP core or SoC architectures.

INTERNET

Website: www.altium.com/tasking
Developer's forum:
www.yahoogroups.com/group/TASKINGforum

RESELLER

Copyright © 2004 Altium Limited.

TASKING, the TASKING logo, Altium and the Altium logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered and unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed. Altium assumes no responsibility for any errors that may appear in this document.

ALTIUM SALES OFFICES

North America

Altium Inc
17140 Bernardo Center Drive, Suite 100
San Diego, CA 92128
Toll Free: 877 TASKING
Tel: 858 521 4280
Fax: 858 485 4610
E-mail: tasking.sales.na@altium.com

Asia – Pacific

Japan – Altium Japan K.K.
Resona Gotanda Building 7F
1-23-9, Nishi-Gotanda
Shinagawa-ku Tokyo 141-0031
Tel: 03 5436 2501
Fax: 03 5436 2505
E-mail: tasking.sales.jp@altium.com

Australia – Altium Limited
Level 14, 39 Murray Street
Hobart TAS 7000
Free Call: 1800 030 949
Fax: 03 6231 4167
E-mail: sales.au@altium.com

Europe

Germany – Altium Europe GmbH
Albert-Nestler-Straße 7
76131 Karlsruhe
Free Call: 0800 0 258486 (0800 0 ALTIUM)
Fax: (0)721 82 44 320
E-mail: tasking.sales.de@altium.com

Switzerland – Protel AG
(A subsidiary of Altium Limited)
Unterdorfstrasse 1
CH-4334 Sisseln
Tel: (0)62 866 41 11
Fax: (0)62 866 41 10
E-mail: tasking.sales.ch@altium.com

From Austria
Free Call: 00800 776 776 77
Free Fax: 00800 776 776 00

From France
Free Call: 0800 88 05 06
Free Fax: 0800 82 85 92

European free call numbers
German speaking: 00800 776 776 77
French speaking: 00800 776 776 55
English speaking: 00800 776 776 44
Free Fax: 00800 776 776 00

1752 TASK DSACPGE-2

Altium Making Electronics Design Easier™