/Diab Compiler: A Long Record of Boosting Embedded Performance

Modern computers only understand ones and zeroes. This can be traced as far back as the first practical digital electronic computer conceptualized around 1937 by John Vincent Atanasoff, a professor at Iowa State College[1].

His prototype tested the two main concepts of storing binary data in capacitors and using electronic logic circuits to perform addition and subtraction. Computers since then have used switches — tube transistors to integrated circuits — to run calculations in the two-state or binary number system.

The ON-OFF Start to Modern Computers

/ **Hardware was the driver behind the binary system as it was easier to reliably** measure and distinguish between the saturated (ON) and cut-off (OFF) states in electronic devices than to measure several states between those values. Additionally, logic processing is quite straightforward in binary.

The binary system, however, does have drawbacks. Firstly, it requires significantly more digits to represent the same thing as a number in the decimal system. For instance, the number 256 in the decimal system is just three digits whereas its binary form, 100000000, requires nine. The number 255 in binary, however, requires one less digit and therefore less storage space to hold that value. This space saving boundary became a constraint felt in early applications, such as 8-bit video games and embedded software, where no number could exceed 255 whether it represented currency in a game or unique colors for graphics.

The other issue with binary is that humans find it is very difficult to read. The simple phrase, “Hello World,” which takes 80 digits in binary, is unrecognizable by most. This makes writing instructions — algorithms and programs — both difficult and time consuming in binary.

The ON-OFF Start to Modern Computers

The difficulty in reading binary was addressed around 1947 with the introduction of low-level human-readable assembly languages that had each of their instructions mapped to corresponding machine language instructions.[2] Higher-level languages (HLLs) soon followed, starting in 1952 with Autocode, the name given to a set of simplified coding systems that were later called programming languages. After that COBOL (1959), C (1970s), and Swift (2014) progressively simplified programming, making it more accessible to humans. They also made programming increasingly efficient with the performance of several complex tasks or functions simultaneously. For instance, a single line of code in C may take several lines of code in Assembly. The overall impact was higher productivity and shorter product times to market.

But how do computers understand anything other than the ones and zeroes? The answer to this question lies in the translation software known as the compiler.

Compilers: How Not to Get Lost in Translation

/ **A compiler is a program that reads each line of a source program in an** HLL and translates or "compiles" it to an object or target language, usually a low-level language or machine-readable code.

While compilers generally support one source programming language, they all have a common structure that addresses compilation in several stages:

1. Lexical analysis: In the first stage, also called scanning, the compiler reads the source code a character at a time, grouping them into tokens that represent building blocks of the program's syntax and may include such objects as keywords, identifiers, operators, punctuation, and constants. The "lexer" or tokenizer thus checks source-code formatting and ensures its output stream of tokens can be processed by the compiler.

2. Syntax analysis: In this stage, the compiler builds a "parse" tree, a hierarchical representation of the program, and uses it to check for syntax errors, such as missing brackets, and reports them for correction.

3. Semantic analysis: The compiler now uses the syntax tree of previous phase and symbol table to check that the source code makes sense, looking for semantic errors like undeclared variables and incorrect function calls or operations using incompatible data types.

Compilers: How Not to Get Lost in Translation

4. Optimization: The fourth stage sees code analysis and optimizations to improve performance. These may include such techniques as constant folding, loop unrolling, and function inlining.

5. Code generation: The parse tree is translated in final stage into machine code executable by the processor. Processor-specific optimizations, like peephole, value following, value numbering, and pipeline scheduling, may be performed at this stage.

Beyond this common structure, developers differentiate compilers by how reliably and quickly they perform the above stages of compilation, what host environments they support, the code size or footprint delivered, and various additional features they offer to help reduce design time, effort, and cost.

Diab Compiler Evolution

/ **The Diab Compiler has continued to evolve for nearly 40 years. Created in** the 1980s, it was architected to have a common compiler for all architectures with architecture-specific details captured in a table file. This helped with both its further development and maintenance and its ability to cross-compile — run on Windows or Linux to generate code for a wide range of CPUs.

The 1990s saw many new processors and embedded microcontrollers, and Diab started supporting Motorola's 68000 (m68k) series, which targeted automotive and similar applications. Soon thereafter, Diab added the reduced instruction set computer (RISC)-based PowerPC, and a long line of processors including Motorola's ColdFire, MIPS, NEC v850, ARM, MCORE, M32R, SH, Infineon TriCore, and Renesas RH850 that were used in mission-critical applications, including automotive, networking, industrial, and aerospace and defense.

Since entering Wind River's stable of products in 2000 through acquisition, Diab has been continuously maintained and, leveraging the company's close relationship with silicon partners, continuously enhanced for new processors with short lead times.

The Diab Toolchain

/ **Diab has evolved into two streams with the 5.9x series continuing to support** a wide range of processors, including ARM. The newer 7.x architecture, however, leverages low-level virtual machine (LLVM) that comprises modular and reusable compiler toolchain technologies. This allows Diab 7.x to stay abreast of evolving embedded needs and SoC architecture advancements, supporting 32-bit and 64-bit Cortex-A, M, and R.

Major components in the Diab 7.x toolchain include:

1. **Driver:** Intelligent program to invoke the compiler, assembler, and linker components

2. **Compiler:** ANSI/ISO C/C++ compatible cross-compiler, which uses LLVM/Clang in 7.0.x and EDG front-end in 5.x versions, and supports ANSI C89, C99, C++03, C++14, and C++17

3. **Assembler:** Macro assembler that generates object modules, supports conditional macros, unlimited number of symbols, and provides information for assembly program debugging

4. **Linker:** Provides precise control of allocation, placement, and alignment of code and data; offers object modules linked into absolute or relocatable modules, and stack usage estimates

The Diab Toolchain

5. GNU Arm® linker support: Two equivalent options are offered to instruct the driver to call the GNU linker for greater GNU compatibility.

6. Libraries: Standard runtime functions. Allows fast, efficient floating-point libraries with full reentrancy; complete C++ library and Standard Template Library (STL); full complement of math libraries, including IEEE-754 appendix functions; library source code

7. Link-time optimization (LTO): Achieves high runtime performance through whole-program analysis and cross-module optimization

8. Undefined Behavior Sanitizer (UBSan): Modifies the program at compilation time to catch undefined behavior.

9. Leak Sanitizer: Identifies runtime memory leaks

10. Instruction set simulator: Simulates the core instructions of the target processor

11. Eclipse CDT plugin: Support for Eclipse plugins and tools for multi-platform embedded development based on GNU toolchains

Key Features, Unique Optimizations

/ **Diab caters to the unique needs of the embedded market with requirement** to pack high performance and functionality into devices with capabilities ranging widely in terms of available memory space and CPU clock frequencies, and power consumption. The compiler therefore offers hundreds of optimization options that enable customizable executables for any embedded environment addressing performance, memory footprint, or both requirements.

Diab's whole-program optimization (WPO) and LTO, mentioned earlier, give developers complete control not only on the performance-code size balance but also on how optimization is achieved. Customers can compile, optimize, and link just one module or extend the process to the entire project at once. During WPO, developers can create groups of modules, for instance, based on mission criticality. This partitioning of modules allows them to customize WPO behavior.

Diab's bag of powerful tool tricks includes the following.

- Small data area optimizer: Predefined sections for widely used static or public variables can optionally be created to improve reference efficiency.

- Code factor optimizer: Diab finds common code sequences at link time and shares them, reducing code size.

- Reverse inlining: This option reduces code size by factoring out repeated code sequences into new functions.

Key Features, Unique Optimizations

- **Easy interrupt handling:** Interrupt keywords and pragmas ease interrupt processing.

- **Position-independent code and data:** Diab can generate code and data that can be loaded at any address. This is useful in devices that dynamically load/unload modules.

- **Control of structure formats:** Packing structures and removing padding reduces footprint. Diab can also create byte-swapped structures to optimize performance when sharing data between big-and little-endian processors.

- **Extensive link command language for memory mapping:** Fine-grained control over optimally laying out code and data in memory addresses the unique memory layouts of embedded devices.

- **Support for multiple object module formats:** The compiler supports ELF, IEEE-695, and S-Records and can generate object modules in multiple formats.

Diab offers many more features that are particularly useful to automotive and industrial applications, including the previously mentioned static stack usage analyzer and UBSan. Other capabilities include floating point runtime hooks for safe handling of floating-point exception related behaviors, allowing developers to trap the exceptions and call appropriate helper functions; memory area cloning feature of the linker to clone a section faster execution; and macro patterns option to improve on the preprocessing time by offering developers wildcard sequences to find and quickly discard code not required.

Code Validation

/ Diab Compiler ships with instruction set simulators – the Wind River WindISS and QEMU – to allow validation of the target architecture. The simulators allow development without the cost toward target hardware, while delivering high productivity during iterative code validation.

When WindISS was developed, it could take advantage of the low CPU clock of most microcontrollers to run robust simulations. While it is a good option for developers to get started with the core instruction set to test their algorithms written in C or C++, the open-source emulator and virtualizer QEMU[3] (derived from Quick Emulator) has come to be preferred.

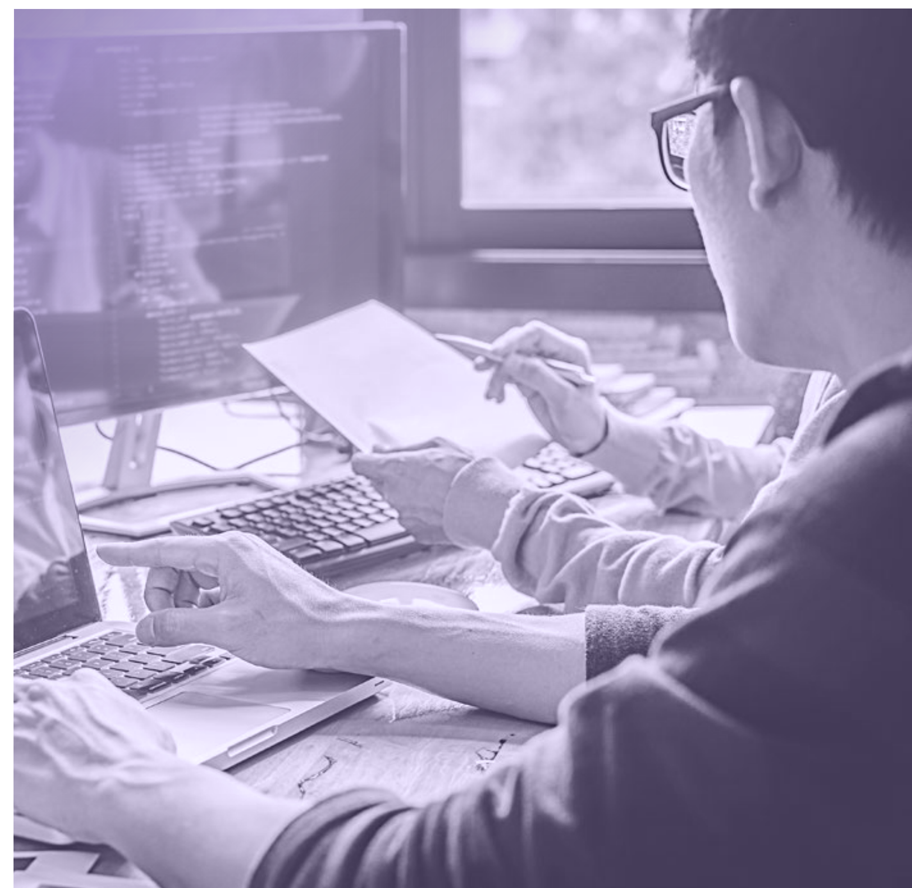
Qemu is compatible with Windows, Linux, and macOS and can run OSes and programs made for, say ARM, on a different machine like a PC. It uses dynamic translation to it achieve high performance, achieving near-native performance by executing the guest code directly on the host CPU.

Safety & Reliability Certified

/ Diab Compiler is certified by TÜV SÜD for developing safety-related software to any ASIL and SIL level. It supports the following safety standards:

- Automotive applications certified to ISO 26262 by TÜV
- Industrial products certified to IEC 61508 by TÜV
- Avionics products certified for DO-178C
- Products for the nuclear market certified to IEC 60880
- Railway applications certified to EN 50128

Diab has been tested with millions of test cases and industry-standard test suites. The compiler ensures interoperability and support for all third-party tools using C and C++ Application Binary Interface (ABI). It uses the industry-standard Itanium ABI for the C++ language as well as standard target-specific embedded ABI (EABI) required when the processor boots to load an application with no intermediate kernel in embedded applications. Diab also includes runtime libraries that conform to POSIX® PSE52, the product standard for OS environments that provide real-time services.



Unmatched Support

/ **Diab Compiler is supported by an award-winning and Service** Capability and Performance (SCP)–certified organization and the Wind River Support Network[4] website that provides patches, manuals, the latest errata, as well as tech tips, application notes, and answers to FAQs. Wind River experts are available for telephone support during standard business hours.

Diab compiler products are supported for a minimum of seven years using the Standard support mechanism to address the long lifecycle nature of embedded systems and for longer with a Legacy support mechanism. In addition, Frozen Branch support is offered for any version beyond the Standard and Legacy support options.[5]

Conclusion

/ Wind River is accelerating digital transformation across industries by delivering the software and expertise that enable the development, deployment, operations, and servicing of mission-critical intelligent systems from the edge to the cloud. The company's technology is found in billions of products and is backed by world-class services and support and a broad partner ecosystem.

Diab Compiler is one such product that supports the transformation of the embedded market, including mission-critical applications in automotive and industrial sectors. Its sustained support for new variants of CPU architectures and introduction of optimizations for ever higher performance and code densities is testimony to its long history of continuous development and market preference.

Learn more at the Wind River Diab Compiler page.

REFERENCES

[1]John Vincent Atanasoff, Britannica

[2]Andrew D. Booth and Kathleen H.V. Britten, Coding for A.R.C., Institute for Advanced Study, Princeton, September 1947

[3]QEMU

[4]Wind River Support Network

[5]Enhanced Support Offerings, Wind River