Tool Demo: Raincode Assembler Compiler

Volodymyr Blagodarov
Raincode, Belgium
vladimir@raincode.com

Ynes Jaradin
Raincode, Belgium
ynes@raincode.com

Vadim Zaytsev
Raincode, Belgium
vadim@grammarware.net

Abstract

IBM's High Level Assembler (HLASM) is a low level programming language for z/Architecture mainframe computers. Many legacy codebases contain large subsets written in HLASM for various reasons, and such components usually had to be manually rewritten in COBOL or PL/I before migration to a modern framework could take place. Now, the Raincode ASM370 compiler for .NET supports HLASM syntax and emulates the data types and behaviour of the original language, allowing one to port, maintain and interactively debug legacy mainframe assembler code under .NET.

ACM Reference Format:
Volodymyr Blagodarov, Ynes Jaradin, and Vadim Zaytsev. 2016. Tool Demo: Raincode Assembler Compiler. In *Proceedings of Proceedings of the Ninth ACM SIGPLAN International Conference on Software Language Engineering (SLE '16).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/2997364.2997387

1 Background

The assembler language for mainframes exists since 1964 when the Basic Assembler Language (BAL) was introduced for the IBM System/360. Around 1970 it was enhanced with macros and extended mnemonics [10] and was shipped on different architectures under the product names Assembler D, Assembler E, Assembler F and Assembler XF. Assembler H's Version 2 became generally available in 1983 after being announced to support an extended architecture in 1981. It was replaced with High Level Assembler in 1992 and subsequently retired with the end of service in 1995. High Level Assembler, or HLASM, survived through six releases: in 1992 (V1R1), 1995 (V1R2), 1998 (V1R3), 2000 (V1R4), 2004 (V1R5), 2013 (V1R6), not counting intermediate updates like adding 64-bit support. It is used in many projects nowadays, mostly for the same reasons the Intel assembler is used in PC applications.

On mainframes, alternatives to HLASM (sometimes referred to as a "*second* generation language" to set it apart from raw machine code) include so-called "*third* generation languages" (3GLs, typically COBOL, PL/I, REXX or CLIST) and "*fourth* generation languages" (4GLs like RPG, CA Gen, PACBASE, Informix/Aubit, ABAP, CSP, QMF — essentially domain-specific languages for report processing, database communication, transaction handling, interfaces, model-based code generation, etc). To name a few concrete examples of good reasons for HLASM usage [14]:

- **Fine-grained error handling**, since it is much easier to circumvent standard error handling mechanisms and (re)define recovery strategies in HLASM than in any 3GL or 4GL.
- **Ad hoc memory management**, since HLASM allows to manipulate addressing modes directly, change them from program to program on the fly, allocate and deallocate storage dynamically.
- **Optimisation** for program size and performance, as well as efficient usage of operating system facilities, not available directly from higher level languages, such as concurrent and reentrant code.
- **Interoperation** of programs compiled for different execution or addressing modes, low-level system access.
- **Tailoring of products**. Many products can be configured or extended by custom user code. However, most of the time, the API is only available as assembler macros.

Additionally, it is not uncommon for a system to be written in assembler in order to evade the costs of a 3GL/4GL compiler, which can be considerable. Such systems are either gradually rewritten to COBOL or PL/I programs, or become legacy. In the latter scenario they can be showstoppers in migration and replatforming projects that can otherwise migrate the remainder of the codebase from mainframe COBOL to one of the desktop COBOL compilers (such as Raincode COBOL) with IDE support, version control, debugging, syntax highlighting, etc. This is the primary business case for developing a compiler for HLASM and the main motivation for us to support it.

2 Problem Description

HLASM is far from being a trivial assembler language: it is possible to use it to represent sequences of machine instructions, but it goes well beyond that. For instance, it helps with idiosyncrasies of the IBM 370 instruction set. In particular, all addresses of memory references have to be represented at the machine level as the content of a register plus a small offset. The assembler can be instructed about what addresses

SLE '16, 31 Oct–1 Nov, 2016, Amsterdam, The Netherlands

the registers contain so that simple references (or complex arithmetic expressions using them) can be converted into a register offset pair automatically. It can also generate sections for literal values that can therefore be written directly in the instruction where a memory reference to the constant is expected.

HLASM is a *macro* assembler with a very rich, Turing complete macro language. The macro and conditional processing is not done as a separate pass and can use the values of already defined symbols that have no forward references. This is extremely powerful for the user but can put a strain on the implementation. The macro processing power is so great that the assembler has been provided with primitives to output simple flat files (rather than relocatable program objects) and several file formats unrelated to machine language (such as CICS BMS maps [25]) are preprocessed with the assembler.

Naturally, the assembler supports all the usual relocations and sections that are expected in modern development environments, and even more exotic features such as multiple entry points, explicit residency mode, merged sections or overlays.

There is a choice of sources that can be used to gain knowledge of HLASM. IBM's *Principle of Operation* [22] provides a good description of the HLASM language and an overview of the z/Architecture. It covers in detail all the instructions and registers provided by HLASM. It also describes how to handle I/O and interrupts. Reading this manual helps understanding the semantics of each instruction. IBM's *General Information* [11] is meant for people already familiar with Assembler H V2, and mostly covers HLASM extensions, as well as the Macro language. IBM's *Programmer Guide* [24] is meant for people familiar with both HLASM and the mainframe environment. It describes how to assemble and run the assembled programs on the mainframe. It also helps in understanding the printed listing produced by the assembler (the listing is the information provided by the mainframe assembler as code comprehension aid). The *Language Reference* [23] has a more practical approach compared to the previously cited IBM's references, and provides a short description of HLASM commands accompanied with code examples.

Several good studybooks were written outside IBM as well [1, 5, 13, 17, 18, 20], mostly either providing shorter (compared to 2000+ pages of official documentation) and more readable versions, or teaching HLASM to beginners by large amounts of code samples. We have mostly relied on Peter Abel's *Programming Assembler Language IBM 370* [1] to understand finer points of assembler programming and execution.

To avoid the strain on the implementation and the subsequent leap of faith for users of the new framework, we have opted for emulating the original architecture as faithfully as possible, including all peculiarities in the use of memory, registers and counters. That way, we can run self-modifying programs and any other typically problematic artefacts and gracefully switch to more optimal and performing alternatives whenever their applicability is guaranteed.

There are 16 general registers: R0, R1, ..., R15. They can be used as base or index in address arithmetic, or as sources and accumulaters in general arithmetic and logic. There are many native operations in HLASM that work on normal binary numbers, binary coded decimals, binary floating point numbers, decimal floating point numbers, hexadecimal floating point numbers, etc, all having their own structure and usages. Remarkably, many operations manipulate data that does not fit within one register and implicitly bundle them in pairs: odd with even for decimal arithmetic and jumping over for floating point. For example, `MR R4,R8` will multiply the values of R5 with R8 and place the result's lower bits in R5 and higher bits in R4. A corresponding floating point instruction `MXBR R4,R8` will multiply a binary floating point number from R4 and R6 with a binary floating point number from R8 and R10. The R0 register cannot be used as a base address or index in addresses, because its code `0000` is used to encode that no base or index is to be applied.

The opcodes of HLASM instructions can be partially decoded (e.g., to see that specific bits represent the instruction length), but in general have to be reimplemented based on a 20-page long table in *Principles of Operation* [22].

3 Solution Architecture

The HLASM compiler is the latest component of the Raincode Stack solution, which already covers PL/I, COBOL, CICS, IMS, DB2, JCL and DF-Sort. The Raincode Stack is a complete mainframe rehosting solution, mainly used to migrate mainframe legacy code to the .NET/Azure platform while preserving its technical dependencies. Most of its components are written either in native .NET languages like C# or F#, or in the metaprogramming languages YAFL [2], DURA [3] or RainCodeScript [4].

When rehosting mainframe applications, it is preferable to keep the code unmodified as it simplifies the migration and guarantees system behaviour preservation, as long as the mainframe behaviour is emulated exactly up to all peculiarities of IBM's compilers. After migration, new development can be done in any .NET language and interact smoothly with legacy applications: we can easily have a PL/I module calling a C# module, which calls the COBOL module, which calls the Visual Basic module, which calls the HLASM one.

To help maintaining legacy applications, Raincode provides a Visual Studio plugin for PL/I, COBOL, JCL and HLASM languages, which offers offers almost the same experience as working with any native .NET language like C#:

- Solution and project management for code organisation;
- Compilation and debugging using the usual Visual Studio IDE menus and shortcuts;

- Advanced editor features such as syntax highlighting, bracket matching, code commenting;
- Visual Studio code snippets for expansion;
- IntelliSense functionality for code completion, click-through navigation, informative hovers, member listings.

In order to support unfortunate but idiomatic assembler constructions such as self-modifying code, executable code defined as data or code fragments used as data constants, we need an appropriate memory representation. First, it has to be faithful of the data manipulated by the program including endianness (mainframes are typically big-endian) and encodings (EBCDIC is the norm). Additionally, the actual assembled machine code must be represented truthfully so that the execution can to react to arbitrary changes to its binary representation. The combination of these factors essentially forces an emulation approach. However, pure emulation has significant drawbacks.

Performance can be inadequate and while self-modifying code can be present, it is quite often limited to very small and specific changes (such as changing the condition on a conditional jump) which could be handled without full emulation of the entire program.

Debugging is not inherently more difficult in an emulator, the main challenge is to find a way to support debugging activities going back and forth between emulated and compiled modules. On the one hand, to be able to debug the assembled code from within Visual Studio in a way consistent with debugging PL/I or Cobol, the DLL (dynamic link library) generated for an assembler module must contain a stepping point corresponding to every instruction. On the other hand, we want to be able to emulate every instruction, at least as a fallback case without sacrificing the debugging ability. Our solution is to have a function in the runtime system for emulating a single instruction per call.

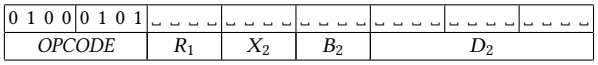
The generated DLL is, by default, mostly a series of calls to that function interspersed with checks that the next instruction to execute is the one coming after (i.e. the instruction just executed was not a jump and did not trigger any exceptional mechanism). This results in the execution playing ping-pong between the generated DLL and the runtime system but gives us the best of both worlds. We can step through the execution, place breakpoints and watches but also have the latest emulated behaviour without having to recompile.

We can also leverage this basic architecture to offer more functionalities. Full tracing at the instruction level can be enabled at assembly time and will result in extra tracing code in the generated DLL but will not impact the performance of other modules. If some module needs more performance than can be provided by a fully emulated solution, the assembler inlines equivalent .NET code for some instructions in the generated DLL. If we want to guard against self-modifying code in this case, we can generate fast checks on the expected memory representation and fall back to the emulator with a very fine granularity of a few instructions.

The main instruction set (basic instructions + extended mnemonics) of HLASM consists of 953 individual instructions, all of them described in the latest release of *Principles of Operation* [22]. Some older instructions, in particular those that existed for IBM 360 and IBM 370, can be also found in various other sources such as the books mentioned above. Each instruction has at least the following properties worth mentioning:

- **Name**, such as "branch and link", intended for human comprehension, **mnemonic**, such as "BAL", intended for programmers and the parser, and **opcode**, such as `0x45`.
- **Format**, defining the bit length of each instruction and the encoding of arguments per nibble.
- **Characteristics**, flags describing frequently occurring behaviour like changing the "condition code" (the result code) for overflows or raising particular kinds of exceptions.
- **Semantics**, described in plain English with the occasional help of examples.

Let us consider the `BAL` instruction in more detail: it has a format "RX-a", which means taking up an entire machine word (4 bytes or 8 nibbles). All RX-a instructions dedicate the first two nibbles to the opcode, which in the case of BAL take the value `0x45` (or `0100 0101`). The next nibble describes R_1, the first argument of the instruction, with four bits representing the register number. The fourth nibble is X_2, the fifth one B_2 and the remaining three are a twelve-bit value of D_2, all three forming the second argument of the instruction as the index, base and displacement of the address in memory:



There are 57 different formats used by HLASM instructions, most described in the documentation in natural language and some erroneously omitted or merged. Each of them had to be coded manually, but this work was necessary and rewarding: for instance, there are 50 instructions using the RX-a format, and the arguments in each one of them are treated uniformly, so the bit arithmetic steps to fetch them are the same and thus can be generated. Some characteristics such as changing the condition code, were utilised in similar ways, but are less structured in the documentation — meaning in practice that the 88 different ways to assign 0, 1, 2 or 3 to the condition code had to be inferred from textual descriptions and then implemented separately in our framework.

After the uniform step of parsing the arguments as addresses, register numbers, relative offsets of immediate values, each instruction has to behave as expected. This semantic had to be coded per instruction, since the original implementation is proprietary and inaccessible, and the correctness of existing emulators [12] is not guaranteed. However, we were able to ease the pain considerably by developing a domain-specific language for expressing typical semantic steps. This allowed us to raise the level of abstraction and specify each operation's semantics in terms of parametrised steps such as:

- Fetch a value from an address in memory
- Extend the sign from an 8-bit value to 32 bits
- Compute an operation on values with a possible overflow
- Assign conditions to possible condition codes
- Convert a value from a zoned to a packed decimal
- Perform an action looping through consequent bytes
- Spread a 64-bit value over bytes in memory
- Set the highest bit of a value to 1

Thus, the emulator function discussed above, can be generated from a sufficiently populated and detailed model of an instruction, and will then fetch the bytes comprising the command, parse the arguments according to the format, perform the computation according to the specified semantic steps, adjust the counters and the condition code, and return successfully to the outer ping-pong loop.

The assembler compiler itself does not contain a definition of the instruction set. All machine instruction are defined in a separate file which is loaded dynamically when the assembler starts. The basic definition of an instruction contains all the necessary information for its parsing: the mnemonic and opcode of the instruction but also information about the arguments of the instruction, their numbers and types (e.g., register, memory reference, immediate) along with the way they are to be serialised in the instruction.

This basic definition can be augmented in several ways. It can contain restrictions on argument values (e.g., some instructions can only be used with an even register). It can give a .NET code template to be inlined for the instruction dependent on the actual values of the arguments [6]. It can describe properties of the instruction semantics (registers accessed implicitly, reading or writing access, jumping instructions, etc.) that can be used by the assembler to produce diagnostic reports and useful visualisations [27] and to optimise the generated code for performance [9, 26].

Because these definitions are external to the assembler executable, they are easy to update in a deployed environment and can be tailored to a particular user. This can be used for hotfixes and workarounds but also to enforce coding guidelines and to address site specific issues (such as dealing with I/O operations).

The last artefact we produce in this project is documentation, which is relatively easy to generate based on the model of each instruction and according to the state of the art practices in software language documentation [30]. We use XHTML + CSS to reconstruct tables and descriptions similar to the ones from *Principles of Operation* [22], which show the characteristics of each instruction, the bit structure of its arguments and the semantic steps both in plain descriptive English and in generated C#. This documentation is a helpful aid in internal knowledge sharing as well as in validating our solution versus the original manuals.

Technically the entire solution is written in C# [7], CIL [8], YAFL [2] and T4 [19]. The HLASM compiler and the Visual Studio plugin were *excluded* from the artefact evaluation due to the conflict of interest (the last co-author co-chairing the AEC), but the information on the product can be found at its official page at https://www.raincode.com/technical-landscape/asm370/.

4 Conclusion

We have reimplemented the IBM High Level Assembler on the .NET Framework. The resulting compiler allows running its instructions and macros in the same environment that already had support for COBOL, PL/I, JCL, CICS and other mainframe languages. It was a technically challenging project with faulty original documentation, multi-phase code generation, model-driven engineering, domain-specific language design, as well as low level optimisations. The Raincode Assembler compiler makes a valuable addition to the Raincode Stack, enabling migration of massive legacy codebases containing large fractions written in BAL, Assembler H or HLASM, from the mainframe platform to .NET/Azure.

The tool we have presented allows one to emulate the execution of HLASM programs directly, compile them to reasonably efficient CIL code, navigate within VS, debug even self-modifying code, and seamlessly integrate them in a .NET ecosystem by performing calls from PL/I, COBOL, C# or VB programs.

There have been some successful projects in migrating HLASM to COBOL or C, far beyond trivial and enough to keep the same team busy for at least two decades: from 1995 [29] to 2013 [28]. In the future we may decide to borrow their experience as well, but not as a part of regular migration strategies. There have also been some attempts at formal verification of assembler programs [16, 21], even though most formalisations are fundamentally inapplicable to self-modifying code, with very rare exceptions [15]. We are looking forward to develop code analysis tools based on those approaches, if the industrial need for them arises.

Acknowledgements

We would like to express appreciation for the work of our colleague Benoît Ragoen who contributed significantly to the project by developing the prototype of the Visual Studio plugin of Raincode Assembler Compiler.

References

[1] P. Abel. *Programming Assembler Language IBM 370.* Prentice Hall, 3rd edition, 1989.

[2] D. Blasband. The YAFL Programming Language. *Journal of Object-Oriented Programming*, 8(7):42–49, 1995.

[3] D. Blasband. Parsing in a Hostile World. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, pages 291–300. IEEE Computer Society, 2001.

[4] D. Blasband. Compilation of Legacy Languages in the 21st Century. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Revised Papers of the 4th International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 7680 of *LNCS*, pages 1–54. Springer, 2011.

[5] F. M. Carrano. *Assembler Language Programming for the IBM 370.* Benjamin-Cummings Pub Co, 1987.

[6] I. J. Davis, M. W. Godfrey, R. C. Holt, S. Mankovski, and N. Minchenko. Analyzing Assembler to Eliminate Dead Functions: An Industrial Experience. In T. Mens, A. Cleve, and R. Ferenc, editors, *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 467–470. IEEE, 2012.

[7] ECMA-334. *C# Language Specification*, 4th edition, June 2006. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[8] ECMA-335. *Common Language Infrastructure (CLI)*, 6th edition, June 2012. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[9] C. J. Fidge. Timing Analysis of Assembler Code Control-Flow Paths. In *Proceedings of the 11th International Symposium of Formal Methods Europe: Getting IT Right (FME)*, volume 2391 of *LNCS*, pages 370–389. Springer, 2002.

[10] GC24-3414-7. *IBM System/360 Disk and Tape Operating Systems Assembler Language.* IBM, 8th edition, Jan. 1970.

[11] GC26-4943-06. *High Level Assembler for z/OS & z/VM & z/VSE Version 1 Release 6 General Information.* IBM, 2013.

[12] D. S. Higgins. PC/370 Release 4.2. http://www.jaymoseley.com/programming/download/pc370v42.zip.

[13] C. J. Kacmar. *IBM 370 Assembly Language With ASSIST. Structured Concepts and Advanced Topics.* Prentice Hall, 1988.

[14] A. F. Kornelis. Why assembler? Bixoft, http://www.bixoft.nl/english/why.htm, 2003.

[15] T. Lake and T. Blanchard. Reverse Engineering of Assembler Programs: A Model-Based Approach and its Logical Basis. In E. Chikofsky, L. Wills, and I. Baxter, editors, *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE)*, pages 67–75. IEEE CS, 1996.

[16] M. Martel. Validation of Assembler Programs for DSPs: a Static Analyzer. In C. Flanagan and A. Zeller, editors, *Proceedings of the 5th Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 8–13. ACM, 2004.

[17] K. McQuillen. *System/360-370 Assembler Language (OS).* Mike Murach, 1975.

[18] K. McQuillen and A. Prince. *MVS Assembler Language.* Mike Murach & Associates, 1987.

[19] MSDN. Code Generation and T4 Text Templates. https://msdn.microsoft.com/en-gb/library/bb126445.aspx.

[20] K. C. O'Kane. *Basic IBM Mainframe Assembly Language Programming.* CreateSpace, 2011.

[21] W. J. Paul, S. Schmaltz, and A. Shadrin. Completing the Automated Verification of a Small Hypervisor — Assembler Code Verification. In M. Holcombe, G. Eleftherakis, and M. Hinchey, editors, *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM)*, volume 7504 of *LNCS*, pages 188–202. Springer, 2012.

[22] SA22-7832-09. *z/Architecture Principles of Operation.* IBM, 10th edition, Sept. 2012.

[23] SC26-4940-06. *High Level Assembler for z/OS & z/VM & z/VSE Version 1 Release 6 Language Reference.* IBM, 2013.

[24] SC26-4941-06. *High Level Assembler for z/OS & z/VM & z/VSE Version 1 Release 6 Programmer's Guide.* IBM, 2013.

[25] SC34-7266-00. *CICS Transaction Server for z/OS Version 5 Release 2 Application Programming Guide SC34.* IBM, 2014.

[26] T. Schüle and K. Schneider. Abstraction of Assembler Programs for Symbolic Worst Case Execution Time Analysis. In S. Malik, L. Fix, and A. B. Kahng, editors, *Proceedings of the 41st Design Automation Conference (DAC)*, pages 107–112. ACM, 2004.

[27] S. Toprak, A. Wichmann, and S. Schupp. Lightweight Structured Visualization of Assembler Control Flow Based on Regular Expressions. In H. Sahraoui, B. Sharif, and A. Zaidman, editors, *Proceedings of the 2nd IEEE Working Conference on Software Visualization (VISSOFT)*, pages 97–106. IEEE Computer Society, 2014.

[28] M. Ward. Assembler Restructuring in FermaT. In J. Vinju, M. Zalewski, J. Rilling, and B. Adams, editors, *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–156. IEEE, 2013.

[29] M. P. Ward and K. H. Bennett. Formal Methods for Legacy Systems. *Journal of Software Maintenance*, 7(3):203–219, 1995.

[30] V. Zaytsev and R. Lämmel. A Unified Format for Language Documents. In B. A. Malloy, S. Staab, and M. van den Brand, editors, *Revised Selected Papers of the 3rd International Conference on Software Language Engineering (SLE)*, volume 6563 of *LNCS*, pages 206–225. Springer, 2010. https://doi.org/10.1007/978-3-642-19440-5_13.

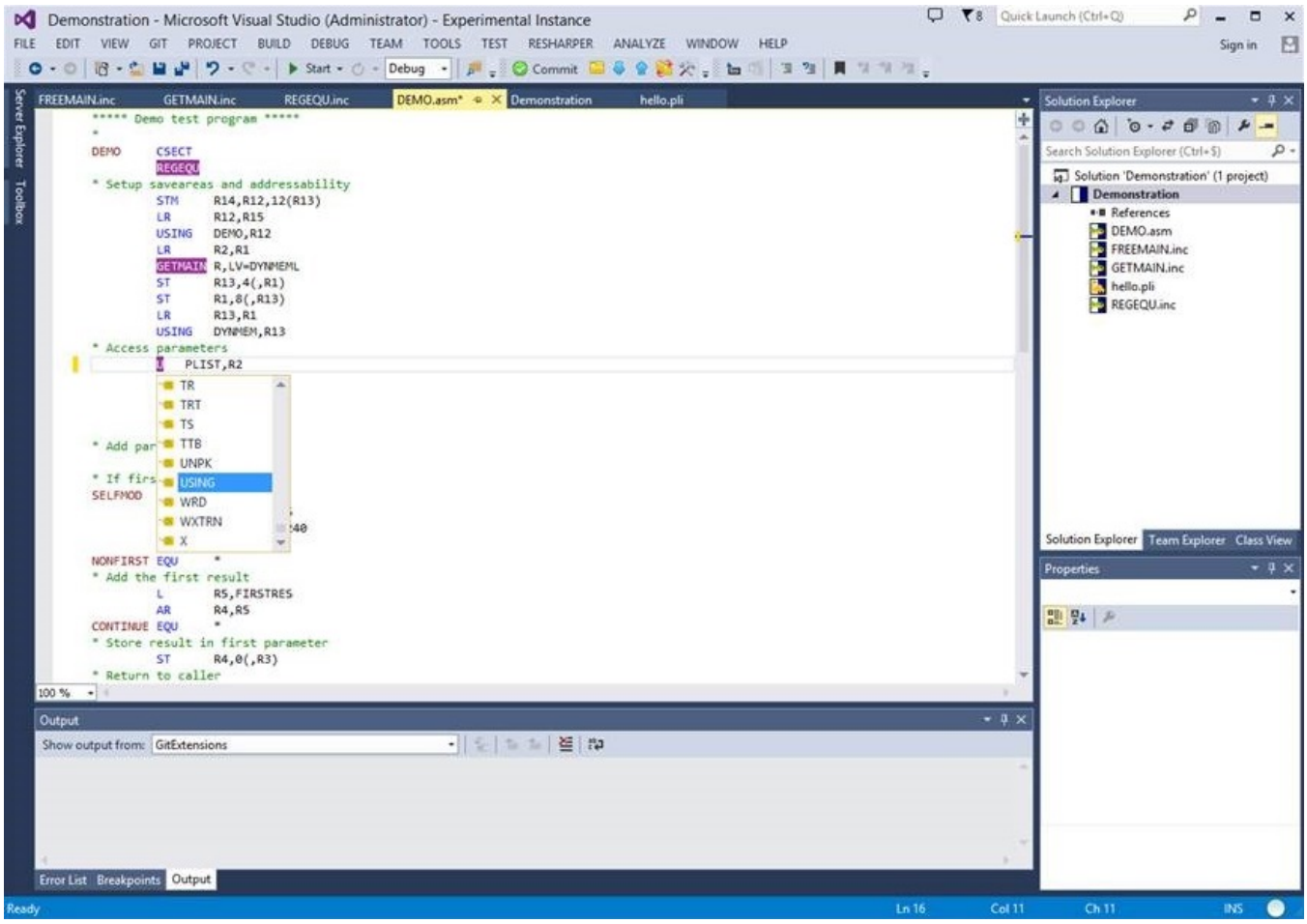


Figure 1. Integration of the Raincode Assembler Compiler into IDE. On the screenshot one can see Visual Studio .NET with a demo solution and a few files open. We also see syntax highlighting and code completion, crucial features of modern IDEs boosting software developers productivity.

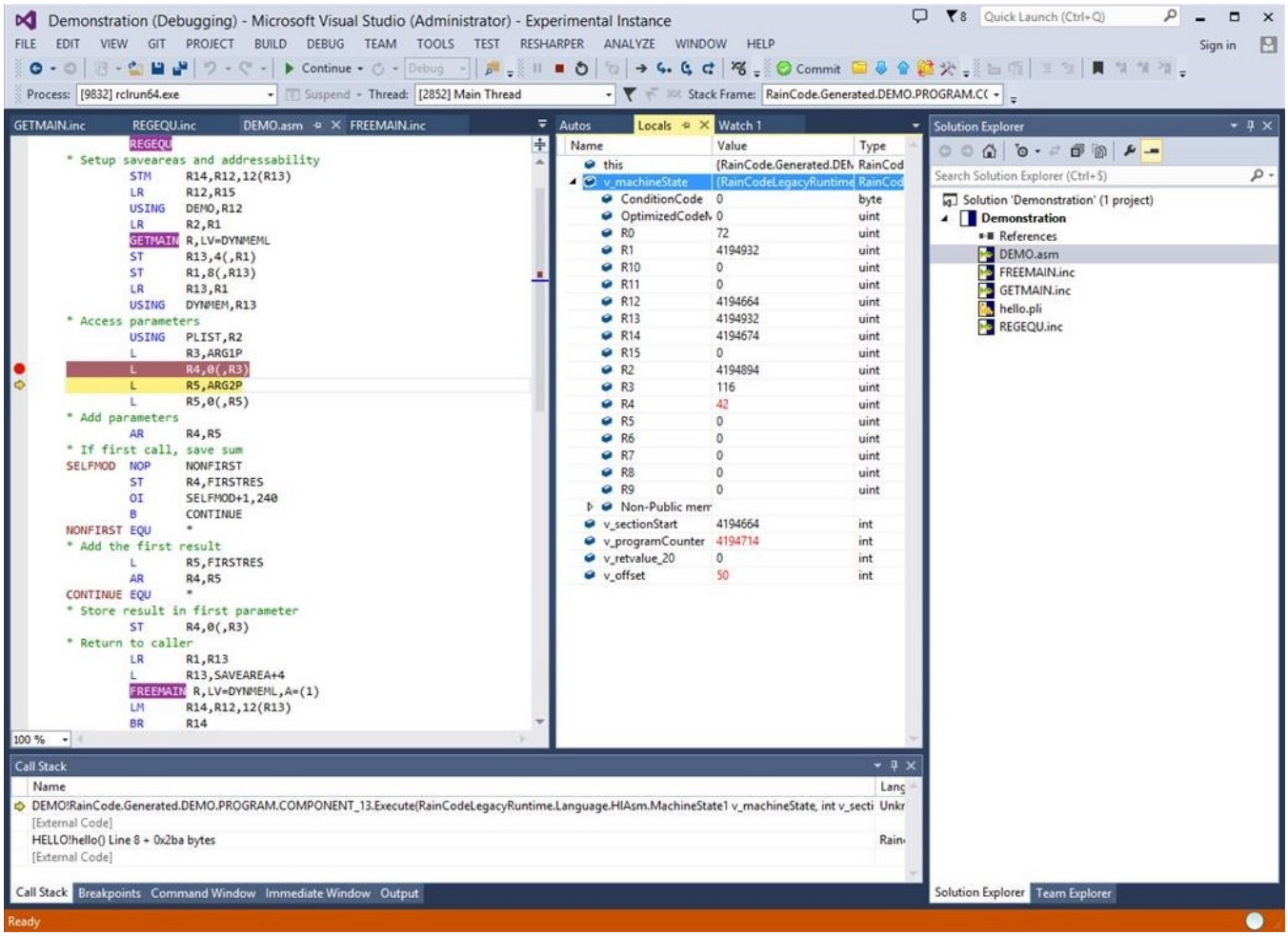


Figure 2. Debugging HLASM programs in Visual Studio .NET with Raincode Assembler Compiler. One can see from the screenshot that it allows to set breakpoints, watch values and step through the code or into the macros. The Raincode Assembler Debugger can cope with self-modifying code as well.