Article

Not peer-reviewed version

@Asm: Augmented Assembly Language

Iosif Iulian Petrila *

Posted Date: 30 January 2026

doi: 10.20944/preprints202601.2364.v1

Keywords: @Asm; assembly language; machine language; augmented language; bootstrappable language; heterogeneous computing

 doi:10.20944/preprints202601.2364.v1

Article

@Asm: Augmented Assembly Language

Iosif Iulian Petrila

Faculty of Automatic Control and Computer Engineering, Gheorghe Asachi Technical University of Iasi, Str. Dimitrie Mangeron, Nr. 27, 700050, Iasi, Romania; iulianpetrila@gmail.com

Abstract

The augmented assembly language @Asm is proposed in order to transcend the fragmentation of architecture-specific dialects, to provide a unified framework for diverse processing paradigms as a universal assembly language and to function as a self-compiling bootstrap instrument adaptable to any processor system. The language augmentations include: flexible machine-language descriptions, general memory and data management directives, custom lexical identification through regular expressions, parsing facilities, generalized macroprocessing, flexible assembly control instructions, customizable encoding and code generation features, compiler-oriented abstraction mechanisms at the language level. The native abstraction augmentations enable expressive and concise high-level descriptions within assembly language for any present, emerging, or future systems.

Keywords: @Asm; assembly language; machine language; augmented language; bootstrappable language; heterogeneous computing

Introduction

Assembly is the only programming language that has accompanied the computing field from its very beginnings to the present day. Emerging alongside the first digital computers [1,2], assembly quickly became the dominant medium of software creation, serving as the primary way operating systems, compilers, and applications were developed. Unlike high-level languages that abstract away hardware details, assembly is essentially machine language slightly abbreviated through mnemonic symbols, replacing raw numeric codes with more human manageable words while preserving a direct correspondence with processor instructions and structures [1–3]. This makes it the truest bridge between hardware and software, where human design is translated into commands that electronic devices can execute. Although today most programming relies on higher-level language abstractions, assembly language, far from being a relic of early computing, endures as the backbone of computing system management being at the confluence between hardware and software, remains indispensable in critical domains where efficiency, precision, and absolute control over system resources are non-negotiable, such as: precise hardware control, embedded systems, firmware and device drivers, operating system kernels, processor-specific runtime environments, performance-critical routines, reasoning about machine-level behavior, instruction set architectures, hardware–software co-design, etc.

Although there are some proposals for architectural uniformization within low or cross level representation [4–9], assembly language as it exists today is fragmented across architectures and platforms, with each ecosystem bound to its own dialect (x86, ARM, RISC-V, WebAssembly, embedded controllers, and more), each reflecting the specific needs and design choices of its hardware [10–14]. This lack of universality not only complicates portability and long-term maintainability but also obscures the underlying principle that all assembly languages ultimately serve the same purpose: to express machine operations in a concise, human-readable form. Alongside with still unresolved issue of unifying conventional processor paradigms (RISC/CISC, register-based/stack-based, etc.), new types of processing systems require a meaningful conceptual integration into a common framework of different computational paradigms (digital, neural,

 doi:10.20944/preprints202601.2364.v1

quantum, analog-redivivus, etc.), as they are currently employed predominantly as specialized and isolated solutions. This highlights the need for a more flexible and extensible low-level language, able of integrating various machine models and computing paradigms [15–17]. Thus, the augmented assembly language integrates the particularities of hardware and provides a common framework for expressing them, ensuring that developers can manage both classical and emerging architectures with the same conceptual tools through general formal translator compiler [6] facilitating powerful abstractions even at the low level, allowing integration into similar augmentations of middle and high level languages [8,9]. Augmentations include multiple aspects such as: machine-level representation and encoding, data management, memory layout, addressing, data definition directives, instruction set architecture, assembly control instructions, macros, lexical, syntactic and semantic analysis tools, etc. Some augmented assembly language specific elements are highlighted in the following.

Augmented Assembly

The proposed augmentations of assembly language cover all language levels, starting from the lowest level, that of machine language, to levels with a high degree of abstraction. These augmentations provide a greater flexibility of assembly language in operating uniformly with different processor architectures through an easier correlation with lower-level languages (machine code, microcode, etc.) but especially with higher-level languages through similar or complementary augmentations with mid-level [8] and high-level [9] languages.

Low-level augmentations involve making machine language operations more flexible to cover the increasing variability of processor architectures through directly including extended machine-level code blocks in assembly language (similar to how the C language allows the inclusion of lower-level code through the **asm** directive), making machine-level representations a first-class concept. Thus, the machine language is augmented with the acceptance of big-endian descriptions, in addition to the usual little-endian descriptions at the hexadecimal byte level separated by spaces, other bases can also be used, including character/ASCII/256 base (which means a value from 0 to 255 that fits in one byte represented by a character within a string type description) as can be seen in the following example related to the x86 processor family (the simplest, complete and most compact possible code of a "Hello, World!" application written in machine language).

```
0100: B4 09           ; mov ah, 9
0102: BA 0108          ; mov dx, M ; M = 0x0108
0105: CD 21           ; int 0x21
0107: C3             ; ret
0108: "Hello, World!" 24  ; M db "Hello, World!", 0x24
```

Beyond compatibility with the conventional disassembly outputs, the augmentations allow parameterization for different execution targets (classical CPUs, virtual machines, neural accelerators, quantum devices, etc.) so that “machine” language/code becomes a generalized, extensible notion rather than a processor-specific artifact. This flexibility allow emerging computational paradigms with weight-encoding or gate-encoding requirements to accommodated through the same directive infrastructure through a machine-agnostic facility, treating their "machine code" as specialized encoding domains rather than fundamentally different abstractions.

Data and code level augmentations related to memory space addressing and management through data / code and memory directives must describe memory regions, alignment constraints, symbolic address spaces, and relocation behavior in a processor-agnostic manner, enabling consistent reasoning about layout regardless of target architecture or assembly dialect. However, the different existing assembler dialects require reconcile syntactic variants to ease adoption and interoperability of some usual legacy directives (**db**, **.byte**, **org**, **.org**, etc.) and treated as syntactic aliases mapped onto a unified semantic core to ensure ecosystem compatibility.

The augmentations at instructions level imply uniformization of common instructions (data movement, arithmetic, control flow, conditional operations, memory access, etc.) with explicitly

 doi:10.20944/preprints202601.2364.v1

architecture-independent declared semantics and virtualization or lowering rules for architecture-specific instructions, allowing the source to express intent at the instruction level while still generating correct, architecture-specific encodings. Thus, the generalized instruction representation provides canonical operation names with well-defined semantics while allowing architecture-specific instantiation. In this way, the system handles operand polymorphism (resolve the tension between architectures with explicit operand ordering as Intel vs. AT&T syntax for instance), implicit operands, and architectural variants through a unified descriptive framework. For example, a general move operation with explicit source/destination operand types can instantiate as **x86** **mov**, **ARM** **mov**/**ldr**, **RISC-V** **mv**/**lw**, or **WASM** **local.get**, depending on operand characteristics and target architecture. The augmentation handles architectural asymmetries through explicit virtualization: architectures lacking certain instruction categories (e.g., RISC-V's absence of complex addressing modes and implicit flags,) trigger automatic expansion into equivalent sequences or alien to classic assembly (e.g. stack-based WASM instructions also managed as general equivalent register-based instructions). These flexibilities will enable a "write-once, tune-everywhere" workflow, bridges the gap between raw machine control and cross-platform portability.

The Abstract Instruction Set Architecture (A-ISA) represents a necessary unifying architectural level in which the augmentations at the instruction categories level are managed unitarily in a more general/abstract processor mode. Thus, these abstract processor elements such as registers and instructions are referenced through logical roles and indices rather than fixed architectural names. For example, a general-purpose register may be uniformly addressed as **r0**, whether it maps to **r0**/**x0** on **ARM**, **ax**/**eax**/**rax** on **x86**, **a0** on **RISC-V**, or a virtual register in **WASM**. This abstraction extends beyond registers to include flags, stack models, and calling conventions, enabling consistent expression of low-level logic while deferring architectural specialization to a later translation phase. This architectural level that capture architectural patterns abstractly allows the uniformization of different processor types in a portable way and also to write assembly code that adapts to different instantiations of similar architectural families (e.g., 32-bit versus 64-bit variants, different SIMD widths, etc.). This level truly achieves machine-language portability because it strictly abstracts existing ISA elements rather than replacing them with higher-level constructs, the present augmented assembly representing exactly a true portable assembly language. By contrast, C language eliminates fundamental machine concepts such as explicit conditional jumps and direct control over execution flow, replacing them with semantic abstractions, which is why C is not a truly portable assembly language, despite often being presented as one.

The metaprogramming level augmentations on assembly language consist on a series of parameterization, composition, and controlled expansion with well-defined compile-time text semantics [6], expressive enough to manage architectural variability, feature selection, and code generation strategies, while remaining predictable and analyzable. These augmentations generalize the usual macros which are simple text substitutions to syntactic and semantic translations through a concept that unites macros, classes, and functions into a more general **class** / **macro** concept [6] which allows explicit expansion rules, parameter evaluation, and conditional logic. These hyper-classes are actually much more flexible macros that can operate over abstract instructions, registers, and architectural features, within text macro-processing. Thus, these facilities will allow easy abstractions at any of the levels presented previously, as can be seen in the following examples of direct explicit definition of some instructions:

```
class mov "x0", "x1"{db 0xE0, 0x03, 0x01, 0xAA} ; ARM64: mov x0, x1
class mov "rax", "rdx"{db 0x48, 0x89, 0xD0}      ; x86-64: mov rax, rdx
class mv "a0", "a1"{db 0x93, 0x05, 0xB5, 0x00}   ; RISC-V: mv a0, a1
class local.get "1"{db 0x20, 0x01}               ; Wasm: local.get 1
```

Along with the power of syntactic and semantic abstraction, metaprogramming augmentations will also allow compile-time (assembly-time) control constructs of the assembler to reason about target capabilities, select appropriate instruction forms, and generate consistent code paths without external processing or code generation tools.

 doi:10.20944/preprints202601.2364.v1

High level augmentations include lexical, syntactic, and semantic analysis mechanisms, enabling it both to bootstrap its own compiler/assembler and also to process high-level structured code according to user-defined syntaxes specific to some high-level language (as for libraries implementations, etc.) as well as the management of natural language descriptions, AI routines directly related to hardware infrastructure, etc. Assembly language obtains in this way a controlled path to higher-level abstractions, without abandoning its low-level foundations, through: pattern matching, tokenization, parsing primitives, code generation, etc. Thus, RegExp (Regular Expressions) can be used in various areas, including in hyper-classes at the argument level to build instruction templates, as in the following example

```
class mov "x(?<rd>\d+)", "x(?<rs>\d+)"          ; ARM64: mov x{rd}, x{rs}
{
  db 0xE0 | rd                                   ; byte 0: rd (bits 0-4)
   db 0x03                                 ; byte 1: fixed encoding
  db rs                                          ; byte 2: rs (bits 0-4)
 db 0xAA                                 ; byte 3: opcode ORR
}
```

where RegExp rules can be even more precise with **(\d|[12]\d|30)** instead of **\d+** or can be supplemented by an assertion **rd < 31 && rs < 31** and a message **"Register out of range"**, etc. Similarly, syntactic and semantic tools can be used within the language, thus facilitating representations that resemble high-level languages and providing a descriptive framework across computational paradigms by incorporate naturally at low-level (where are there major differences) non-traditional architectures, including neural, AI-based systems, quantum models, etc.

Although historically assembly language introduced the first level of abstraction over machine language, it has remained indispensable for the precise description of processor-level instructions in critical areas of programming instruments. Assembly language continues to be the point at which executable code debugging, reverse engineering and disassembly-based decompilation processes typically terminate, as up to assembly level translations to/from machine language can still be considered reversible. Assembly augmentations enable direct, powerful and concise abstractions that closely reflect the underlying execution model, allowing explicit control over registers, memory layout, instruction sequencing, and processor-specific features that are often inaccessible or deliberately hidden in high-level languages. This direct expressiveness makes assembly uniquely suited for defining a universal, architecture-aware intermediate representation, capable of faithfully modeling execution semantics while remaining close enough to hardware to expose optimization opportunities, non-standard instructions, and specialized execution patterns that cannot be reliably expressed elsewhere in higher level languages which have already become inefficient in integrating different architectures without implementation critical libraries in lower-level languages.

Conclusions

The augmented assembly language introduces a set of low-level features such as: flexible machine language descriptions, abstracted instruction set architectures, memory management through architecture-independent directives for space reservation, layout control, addressing semantics, and allowing uniform treatment of diverse memory models.

The augmented assembly language introduces a set of higher-level descriptive constructs, such as: pattern definitions, token-based matching, user-defined parsing, and code generation facilities, explicitly designed to build upon lower-level assembly primitives, allowing different present or emerging architectures (neural, quantum, natural language models) to be described using the same formal framework, right at assembly level where the architectural differences can be easily and concise uniformed and abstracted.

The augmented assembly language integrates compiler-oriented abstraction mechanisms at the language level that enables the language to operate simultaneously as an assembler, a programmable intermediate representation, and a transformation framework. The resulting system reduces dialect

 doi:10.20944/preprints202601.2364.v1

fragmentation while preserving low-level expressiveness and control, providing an immediately usable toolchain for targeting classical processors as well as emerging computational units, positioning the language as a unifying substrate for implementing libraries of other programming languages but also to function as a bootstrapping language easily adaptable to any type of processing unit.

References

1. H. H. Goldstine, J. Von Neumann, *Planning and Coding of Problems for an Electronic Computing Instrument*, Institute for Advanced Study, Princeton (1947).
2. A. Booth, K. Britten, *Coding for A.R.C.*, Institute for Advanced Study, Princeton (1947).
3. M.V. Wilkes, D. Wheeler, S. Gill, *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley (1951).
4. C. Lattner, V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, Proceedings of the International Symposium on Code Generation and Optimization, 75-86 (2004).
5. W. Hagen, *The Definitive Guide to GCC*, Apress (2006).
6. I. I. Petrila, *Implementation of general formal translators*, arXiv:2212.08482 (2022).
7. S. V. Vangavolu, *Universal Development with WASI: Building Secure Cross-Platform Apps Using WebAssembly System Interface*, European Journal of Computer Science and Information Technology 13 (2025) 1-14.
8. I. I. Petrila, *@C – augmented version of C programming language*, arXiv:2212.11245 (2022).
9. I. I. Petrila, *@JavaScript: Augmented JavaScript*, Preprints 202502.0081 (2025).
10. Intel Corporation, *The 8086 Family User's Manual*, Intel Corporation (1978).
11. G. Kane, J. Heinrich, *MIPS RISC Architecture*, Prentice Hall (1992).
12. S. B. Furber, *ARM System Architecture*, Addison-Wesley (1996).
13. K. Asanovic, D. A. Patterson, *The RISC-V Instruction Set Manual*, Berkeley (2014).
14. A. Rossberg, *WebAssembly Specification*, W3C (2018).
15. I. I. Petrila, *Neural Information Organizing and Processing Principles*, Preprints 202502.0827 (2025).
16. C. Yuan, A. Villanyi, M. Carbin, *Quantum control machine: The limits of control flow in quantum programming*, Proceedings of the ACM on Programming Languages 8 (2024) 1-28.
17. X. Liang, J. Tang, Y. Zhong, B. Gao, H. Qian, H. Wu, *Physical reservoir computing with emerging electronics*, Nature Electronics 7 (2024) 193-206.