*Enthusi*ASM: A Tile-Based Visual Programming Language for High-Level Assembly

Matthew D. Boardman
Dalhousie University
Matt.Boardman@dal.ca

Abstract

In this paper, a visual programming environment for the development of low-level and high-level assembly language programs is proposed, and is evaluated in the context of Green and Petre's "Cognitive Dimensions" framework. The system acts as a code generator for a textual assembly compiler, and is designed to be flexible enough to accommodate any hardware architecture and operating system. As an implementation example, an interface prototype for the common 32-bit Intel Architecture is shown in this paper. In addition to low-level assembly instructions unique to each hardware platform, the language includes support for high-level assembly language programming structures such as conditional constructs, looping constructs and stack management.

1. Introduction

*Enthusi*ASM is a realistic proposal for a Visual Programming Language (VPL) designed for the development of both low-level and high-level assembly language programs. The proposed system is primarily targeted towards education: students who wish to gain a greater understanding of the inner workings of modern processors without regard to the arbitrary syntactical demands of assembly compilers and varied addressing schemes, such as compiler-specific directives, or the notation for immediate, direct or indirect addressing modes. The system is also designed to simplify interaction with external code modules, so that programmers used to high-level languages, who use assembly language infrequently, may quickly and simply design high-performance assembly modules for use in applications that have the majority of their code written in compiled high-level languages such as C or C++. To alleviate tedious, repetitive tasks in assembly programming, support for high-level structures are provided, such as *if-then* or *case* conditional statements, *do-while* or *repeat-until* loops, and automated stack management for the invocation of local procedures, procedures from object files in other languages, an assembly compiler's standard library procedures or operating system-specific Application Programming Interface (API) functions to allow access to system hardware via the kernel. Support for the API also means that full Graphical User Interfaces (GUI) may be designed entirely in assembly, by accessing standard GUI development tools and including resources during compilation.

*Enthusi*ASM is both a VPL and an Integrated Development Environment (IDE). It is designed to be flexible enough to be applied to any hardware architecture, assembly language compiler or operating system by simply changing its configuration. In this paper, as an implementation example, the common Intel Architecture 32-bit standard (IA-32) will be targeted (more specifically, the Intel Pentium Pro or x686 architecture) [10], using the MASM32 assembly compiler [12] with the Microsoft Windows operating system [13]. Although it would be natural for *Enthusi*ASM to be implemented in Java for portability, in Plate 1 the prototype interface is shown using Microsoft Windows-specific controls for convenience.

1.1. Assembly Language

Today's general-purpose, stored-program computers use binary data to store both data and application code. All high-level languages are converted at some point to binary code for low-level execution at the machine level. Some high-level languages such as C or C++ are converted directly into machine language through compilation, while interpreted languages such as Java execute through an intermediary: in Java this intermediate layer is called the Java Virtual Machine (JVM). The many binary instructions, or opcodes, making up this machine code vary by processor family, and are difficult for human programmers to remember. Depending on the addressing mode or length of immedi-

ate values (constants) used in the instruction, the binary opcode may change in value. Complicating the matter further is the fact that in some architectures, opcodes may be of variable length: for example, in IA-32, an opcode may be one, two or three bytes in length [10], in addition to further bytes needed to identify specific memory locations or immediate values, all for a single instruction.

Assembly language was therefore created for convenience, to translate each opcode into a human-readable symbolic interpretation. For example, in IA-32 machine code, a simple instruction to add three to the 16-bit register AX might appear as [10]

```
05 03 00
```

The first value specifies the "add 16-bit immediate to AX register" opcode, and the next two values specify the 16-bit constant. Note that here these values are in hexadecimal for readability, and that since this example uses the IA-32 standard, the immediate value appears as little-endian, i.e. least significant byte first. However, in symbolic assembly language, this same instruction would be specified as

```
add AX,3
```

The symbolic interpretation, or *mnemonic*, of the machine code for this instruction is `add`, and since immediate addressing is used and the register to employ in the addition is specified, the assembly language compiler knows to convert this instruction to the `05` machine code. Note that the order of the `AX` and `3` arguments, or *operands*, for this instruction is *destination* followed by *source*, since we are using an Intel processor: in other standards such as the Motorola 68000 processor family, this order would be reversed [1]. This non-standardized ordering of operands is another complication for novice assembly language programmers, who must not only deal with the syntactical idiosyncrasies of each assembly language specific to a particular processor family or assembly compiler, but also the often arcane addressing specifications, and any memory model architectures specific to a particular operating system.

1.2. Modern Assembly Languages

In addition to the syntactical and addressing difficulties for novice programmers mentioned above, the complexities of CISC (Complex Instruction Set Computing) processors compound the problem by providing an intimidating plethora of available instructions [1, 3]. Programmers must also consider the byte-order of stored simple data elements—whether a processor is little-endian as for Intel processors or big-endian as for Motorola processors—which complicates matters for programmers switching from one hardware architecture to another. Generally speaking, reading and writing assembly programs is prone to error from many directions, complicating maintenance efforts [18]. High-level languages such as C or C++ are compiled directly into machine code: modern compilers are quite adept at writing efficient machine code, and are able to consider many more factors than most human programmers, such as out-of-order execution[16], speculative execution of conditional branches[2], multiple, deep instruction pipelines [11] and advanced cache mechanisms [16]. So why is assembly necessary?

Other than those programmers who actually develop the compilers, it is useful for any student of modern computer languages to understand assembly language, in order to better understand the execution environment of the underlying hardware architecture [1, 3]. Execution performance and smaller code size have been raised as good reasons to program in assembly [8, 18], and in limited environments such as for use with embedded devices [18] or for time-critical kernel operations, these may be important factors.

It is possible that an expert assembly language programmer may always be able to create better code than even the best compilers, simply because the human programmer can always look at the output of the compiler and make small, fine-grained adjustments to "tweak" the result [8]. For this reason, it is quite conceivable that an application with intensive data processing, such as data encryption and compression, or that moves large amounts of data between hardware devices, as with multimedia, might have the vast majority of its code created in a high-level language but have specific modules of code written in assembly language in order to achieve the highest possible level of performance [2, 8, 9]. Also, it is well-known that many virus writers use assembly language to create the smallest possible code to "infect" valid executables in order to avoid detection.

For applications with a greater proportion of time-critical code, it may be necessary to write the majority of application code in assembly. Indeed, recent advances in high-level assembly compilers allow programmers to provide a complete GUI through system calls to the operating system [9], such as MASM32 and FASM for Microsoft Windows [5, 12], or NASM for Linux [19]. High-level assembly language provides a further level of abstraction from the hardware architecture, in which high-level macros, or pre-defined sequences of instructions, automate common tasks such as the creation of conditional statements, loop structures or the invocation of kernel API. These are provided for code readability and convenience, but since each macro

translates directly to a series of assembly instructions, they are still architecture-dependent.

IDEs for these modern assembly compilers allow fully developed GUI to be created entirely within the confines of assembly, with roughly the same level of difficulty as programming such an application in C [9], by using high-level macros to call operating system-specific API and by employing standard memory models, such as 32-bit protected mode in an Intel environment and the Microsoft Windows-specific Portable Executable (PE) standard [9] for the format of executable programs, to free the developer from the concerns of memory segmentation.

1.3. The Intel x686 "Pentium Pro"

In this paper, the proposed *Enthusi*ASM implementation will consider the Intel x686 processor architecture, as this the most common Intel standard currently in use.

Intel's development of the "Pentium Pro" processor in 1995 represented a significant departure from the purely CISC architecture which Intel had used in the past [16]. In order to maintain binary compatibility with the Intel family, the processor needed to interpret the legacy Intel Architecture (IA) opcodes. However, in order to take advantage of recent Very Large Scale Integration (VLSI) hardware advances, a RISC (Reduced Instruction Set Computing) architecture with a fixed-length opcode were needed. Intel reconciled these conflicting goals by adding translator hardware to parse each IA-32 opcode into a series of fixed-length *micro-ops* [16], which then execute on a RISC platform. In addition, the restrictive set of eight General Purpose Registers (GPR) in the IA-32 standard were replaced with forty, 32-bit multipurpose GPRs for use by these micro-op instructions [11, 16].

The processor allows a 32-bit addressing for a maximum of 4 GB of addressable memory space per process, but 36-bit physical memory addressing for 64 GB of physical address space [11, 16]. Few additional IA-32 instructions were added with the first Pentium Pro processor, but subsequent Intel processors using this same CISC to RISC translation scheme, such as the Pentium II through Pentium IV, have added additional instruction sets and register sets such as the MMX instructions and registers used for multimedia applications [10].

1.4. Visual Assembly Programming

Many IDEs for assembly languages claim to be visual, yet do not incorporate VPL methodologies such as a data-driven execution model [17] or visual algorithm creation [4, 20]. Such IDEs are generally limited to coloured syntax highlighting or the inclusion of resource editors for drawing icons and creating GUIs [4].

Genuine VPLs with graphical methods for building algorithms have been proposed and several have been commercially successful, such as LabVIEW from National Instruments, designed for use by scientists and engineers with a hardware connection paradigm [4, 20], and Prograph from Pictorius, Inc., which was designed as a general-purpose visual language employing object-oriented concepts [6]. However, many of these VPLs are specific to a particular problem domain, and commonly employ a data flow model in which dependencies within the data guide the execution path [17]. In the case of general-purpose, low-level assembly programming, which is naturally a control-flow language consisting of a sequence of imperative commands [17], it is difficult to imagine how a data flow model might be employed.

Tile-based VPLs have been proposed such as AgentSheets [15], in which autonomous Agents, in a rough parallel to class objects in object-oriented design, are represented by graphical tiles which can be manipulated by the programmer as building blocks to create graphical simulations of real-world events. Visual processor simulations have been proposed, such as [1, 14] for the Motorola 68000 processor or [3] for early Intel processors, in order to visualize the execution of assembly instructions for educational purposes. The simulation proposed in [1] consists of "templates," unique to each assembly instruction, which are filled in to provide a functional processor instruction to be visualized. Visual assembly language editors have also been proposed, such as EasyAssembly [18] for the MIPS mainframe architecture, which allows instructions to be created by dragging registers to memory or stack locations in a graphical environment.

2. *Enthusi*ASM

The proposed *Enthusi*ASM system combines elements from several of these approaches. Programs are created by building a vertical series of fixed-height tiles, executed from top to bottom, to maintain sequential, imperative control flow. Each tile may represent a machine instruction, a high-level macro consisting of a group of several instructions, a pre-defined conditional or iterative construct, or an invocation of another procedure. Larger conditional and iterative constructs may include several tiles: for example, a *repeat-until* loop has an initial tile indicating the start of the repeat structure and an ending tile indicating the condition for termination.

Colour is an integral component of these tiles to quickly convey meaning: green tiles are native processor instructions, yellow tiles indicate a high-level macro or an invocation tile, whilst blue tiles indicate conditional or iterative high-level constructs. In a practical implementation, it would be advisable to allow these colours to be modified by the user to account for colour-blindness or low contrast displays.

For example, a common `mov` instruction to copy data from a register to a memory location might appear in machine language as

```
mov Line_No,EAX
```

To create this instruction in *Enthusi*ASM, a developer would drag-and-drop a `mov` tile from a list of available tiles to the needed location in the current list of tiles, then drag a register location from a window depicting available registers within the current processor architecture, and finally drag a memory value from a list of pre-defined variable locations for the current project.

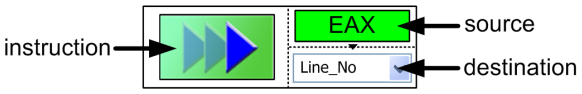

Figure 1. A `mov` instruction tile.

The result of these actions is shown in Fig. 1, with three drag-and-drop locations (an instruction to the left, a source at the upper right, and a destination at the lower right) separated by dotted lines. A small arrow pointing downwards reminds the user which location is the source and which is the destination: this allows additional abstraction from the peculiarities of a specific assembly compiler syntax, as the tile would appear identical for Intel notation and Motorola notation even though in the textual syntax the order of the operands would be reversed [1].

The conceptual compilation architecture of *Enthusi*ASM is shown in Fig. 2. A project may consist of one or several assembly files, each of which may contain one or several procedures. In order to generate the proper textual syntax for the required assembly compiler, *Enthusi*ASM needs to know the specifications of the processor, such as the available instructions and addressing modes, and the segmented memory model used by the particular operating system. The system also needs a series of pre-defined macro tiles specific to each processor, operating system and/or assembly compiler. During compilation, *Enthusi*ASM generates the required assembly files from the user-created sequential lists of tiles, in the syntax of the configured assembly compiler, which then creates binary object files. These

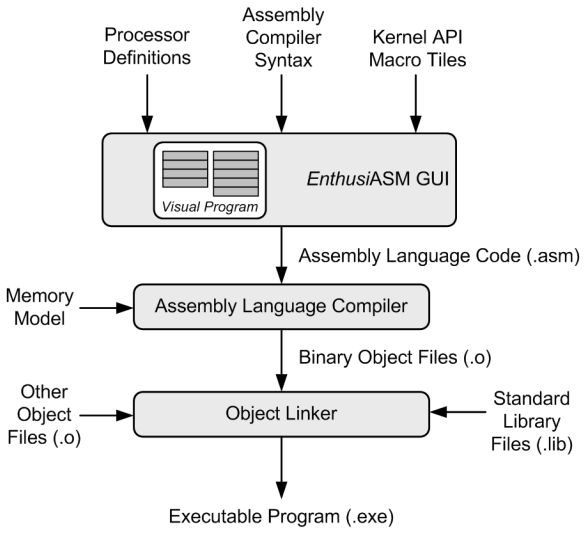

Figure 2. The *Enthusi*ASM compilation architecture.

files may be linked with other binary object files created by higher level languages such as C, or with standard library files specific to a particular operating system such as math libraries and kernel APIs. The result of this compilation of a project is a single executable program.

2.1. Graphical User Interface

An example of the proposed *Enthusi*ASM GUI is shown in Plate 1. The user is currently dragging a tile to create a `mov` memory instruction (dashed red line at lower left). All windows are resizable according to the user's preference, and may be hidden or positioned anywhere within the main window.

In this figure, **Code Viewer** and **Tiles** windows show visual and textual views of the same assembly procedure *hello*, in the *hello.asm* assembly file, for the *hello.prj* project. The text windows uses coloured syntax highlighting for readability. The user may not modify its contents, however, since the contents of the window are kept in synch with instructions created in the *Tiles* window, comparing these two views enables the novice assembly programmer to learn the instruction mnemonics and addressing modes used by the processor architecture, as well as the syntax of the assembly compiler. This aids in presenting a gentle learning curve for novice users.

The **Navigator** window orients the user by showing that the current project contains a single assembly file *hello.asm*, with two procedures, *main* and *hello*. Dotted lines in this **Navigator** show which procedures are in which file, while solid arrows show a simple call-

ing hierarchy to provide a view of program structure. The user is free to move these boxes anywhere within the **Navigator** window; the arrows and dotted lines are moved with them automatically. If a line crosses a box, the line is drawn on top of the box, to aid visibility. Buttons along the right edge allow the user to zoom in and out, and change the colour assignment for each subroutine at their own discretion. For example, the user may choose to colour procedures for the user interface in green and information processing procedures in blue, or may choose a colour based on the level of testing performed. By clicking on a particular assembly file in the **Navigator** window, the user is presented with a list of options for that particular file, as shown in the figure, such as the memory model or procedure calling convention: although these options have defaults in all cases, it may become necessary for a user to change one of these values (for example, to include an additional library).

A hierarchical list of pre-defined tiles for this architecture is shown in **Available Tiles** window, providing a convenient organization for quickly locating any given instruction so that the user can drag-and-drop a tile to the *Tiles* window to create a new instruction. Within each grouping the mnemonics are presented in alphabetical order, and a search bar is provided to find a specific, known mnemonic quickly. This hierarchical approach is particularly needed for CISC processors, which may have an enormous list of available instructions: for example, the software reference manual listing instructions for the Intel processor family (as of this writing) comprises two volumes and totals over 1200 pages [10]. For this architecture, the memory instructions such as Load, Store and Move are presented first, followed by instructions for Integer and Floating Point operations and so on. Following these processor-specific instruction tiles, conditional and iterative tile structures are shown. Next we see high-level macros—tiles with a group of pre-defined processor instructions to automate repetitive tasks—defined in either global scope (available to all projects) or local scope (available to this project only). Finally, we see the operating system-specific tiles for API invocation.

The **Processor** window shows the available register locations for the particular processor, organized in a manner logical to the particular architecture using tabbed canvases. In IA-32, for example, the GPRs are shown on the first tabbed canvas. The status registers, EIP (the instruction pointer, or program counter) and EFLAGS (showing single-bit flags such as Overflow, Carry and Interrupt [16]), are shown on the second canvas. Other less-used register groupings, such as segment, debug and control registers, are shown on subsequent canvases. The eight, 80-bit Floating Point Unit (FPU) data registers and six control registers [10] also require their own canvases, and further canvases can be defined for advanced features such as MMX registers. These are provided so that the user may drag a specific register to the **Tiles** window while creating an instruction in a similar manner to [18], and also provide additional functionality during debugging.

The **Memory** window shows the list of local (near) variables and constants that have been defined by the user within the current procedure's data segment, or vector (far) variables for use with indirect addressing in another segment [2], which may be dynamically allocated such as arrays whose size may be unknown until runtime. On other tabbed canvases in this window, we see the input and output stacks that have been defined for the current procedure: defining variables in these areas provides automated stack management, so that a programmer no longer needs to manually specify `push` and `pop` instructions to fill and empty the memory stack manually before and after each procedure call. This also ensures that each argument has the correct length in bytes. The size of each variable or constant is shown in both the number of bytes and in the lingo of the particular architecture, for example BYTE, WORD, DWORD or QWORD for 8, 16, 32 or 64-bit integers; SZ for a zero-terminated array of characters; or DWORD, QWORD or TWORD for 32, 64 or 80-bit floating point variables.

Note that the user is completely isolated from any syntactical requirements for the addressing modes of the particular architecture and assembly compiler, and as a direct result of this the programs are much less prone to error. No provision for higher level structures (such as would be created as class data in C++ or a `struct` in C) is included, other than reserving space and providing a dynamic pointer to the top of such structures: the developer would be responsible for creating local macros which perform the input and output for such customized structures. In this way, for example, a C programmer might combine C modules to manipulate complex data structures with assembly modules to process a high volume of simpler data elements. As with the **Processor** window, the user may drag a variable or constant to the **Tiles** window to specify the source or destination for an instruction.

2.2. Program Structure

In Plate 1, note the tiles included in the **Tiles** window defining the *hello* procedure. From the top, we have a "start" label, a `mov` memory instruction tile and an incomplete `lea` load instruction tile surrounded in red (to indicate that the tile cannot be used in debugging

or compilation). Note that in *Enthusi*ASM, the graphical icon for each instruction doesn't have to necessarily be unique, as this would require the user to memorize hundreds of icons rather than hundreds of mnemonics. Rather, the icon indicates a more general category of mnemonic. For example, in the **Available Tiles** window in Plate 1, the list includes two load instructions that are associated with the same graphical icon. By hovering the mouse over any graphical icon, the user is shown the exact mnemonic in a popup help box.

These are followed by a yellow macro tile with a kernel invocation (the *MessageBox* procedure from the *User32* kernel API group [9]). The procedure to call in the invocation may be specified either by dragging an appropriate invocation tile from the list, or by dragging an empty invocation tile and typing the name of the procedure. If the procedure does not yet exist in any scope, a new, empty procedure will be created in the current project. Right-clicking on an invocation tile shows a popup menu, from which the user can choose to modify the input and output stacks for the invocation.

Next, an `inc` integer instruction tile is shown, which adds one to the AX register (note that the second operand is unnecessary in this case and therefore is disabled). This is followed by a simple conditional structure: all macro tiles for conditional and iterative structures begin with a blue tile, with a graphical icon specifying its function. In this case, the question icon shows that this is a conditional statement, and the inclusion of the condition in the initial tile indicates that this is an *if-then-else* structure. In this case, if the AX register is greater than the value of the COUNT memory location, an `add` instruction and a `mov` instruction will be executed, otherwise a `jmp` unconditional jump instruction will be executed. Note that nested structures are quite legal within this syntax, with the indentation in these high-level constructs entirely controlled by the GUI (although if only low-level instructions are used with no high-level tiles, no indentation is necessary). A *case* structure would be similar, but would have only the variable or register to be tested in the initial macro tile, additional tiles between each case to indicate this case's match value, and a final *otherwise* case.

We then see another `mov` memory operation, followed by a looping construct. Since the condition for the looping construct is shown in the initial tile, this is interpreted as a *do-while* loop. A *repeat-until* loop would look similar, but would specify no condition in the initial macro tile: an additional loop tile at the end of the loop would instead show the condition, with a stop icon to indicate that this is the stopping condition. Finally, we see an empty location into which the user is currently dragging a `mov` instruction from the list of available tiles. Although not shown in the figure to save space, textual comments can be placed anywhere in the tile diagram using a right-click. These comments then appear in the generated assembly code, as shown in the **Code Viewer** window in the figure.

2.3. Debugging Environment

Arbitrary constraints imposed by the processor architecture are followed by checking the legality of each instruction in the context of the chosen assembly compiler upon the completion of each instruction. For example, the IA-32 `mov` instruction takes two operands: a source and a destination. The source is allowed to be an immediate value such as a constant, a register or a memory location. The destination is allowed to be a register or memory location. However, for historical reasons, only one access to memory is allowed per instruction. Although this restriction would seem unnecessary in the Pentium Pro architecture, since all instructions are converted to micro-op instructions prior to execution [16], this restriction is maintained in the translation engine of the processor and therefore must be enforced during assembly language compilation. In *Enthusi*ASM, if the developer creates a `mov` instruction with both the source and destination specified as variables, the operation will be surrounded by a red box to indicate an error. By hovering the mouse over such an instruction, the user may be informed of the particular error in a popup help box.

During development, a procedure may be called at any time without compilation. This capability is provided by a simulation layer, in which each processor instruction corresponds to a particular action within the simulator, in a similar manner to [1, 14]. Reverse execution of simple instructions is carried out by remembering what has changed after each instruction is executed, in an "undo"-like manner, but is limited to instruction tiles (not macro tiles). This provides a progressive execution environment, in a similar manner to Prograph [4, 6]. While the program is "paused" at a breakpoint or during step-wise execution, the user is free to insert and remove tiles both before and after the current instruction, or modify the contents of the registers. Enabled breakpoints are shown with a red stop icon, and temporarily disabled breakpoints are shown with a yellow exclamation icon. A green caret between the add instruction and the conditional structure in the *Tiles* window of Plate 1 shows where the current execution has paused: this caret is present in both the **Code Viewer** and **Tiles** views.

Fig. 3 shows the **Debug Toolbar**, which appears to the right of standard toolbar buttons such as cut, copy and paste as shown in Plate 1. From the left,

these buttons are *Stop* (enabled during continuous execution), *Step Backwards* (for step-wise reverse execution), *Step Forwards* (for step-wise forward execution), *Run to Next Breakpoint* and *Run Continuously*. This last option does not use the instruction simulation layer, but rather runs natively on the processor hardware; it is enabled only when the current architecture matches the configured target architecture.

Figure 3. *Enthusi*ASM's Debug Toolbar.

During debugging, each register location in the **Processor** window shows its current value, and is shown in grey if it has not yet been modified within the current procedure. A **Stack Watcher** window informs the user of the current call stack (which procedures in which files have been called in what order) and the current memory stack created by `push` and `pop` operations or by procedure invocation macro tiles.

3. Discussion

The "Cognitive Dimensions" framework proposed by Green and Petre [4] provide a basis for the evaluation of VPLs. Here, we will compare *Enthusi*ASM to textual assembly languages in this context, assuming familiarity with these dimensions from the reader.

By freeing the user from syntactical trivialities such as the ordering of operands and various addressing modes, *Enthusi*ASM provides a greater detachment from the hardware, offering a great advantage in the *Error Proneness* dimension: this is furthered by providing a simple drag-and-drop interface to virtually eliminate typing errors. Since the same visual syntax is used regardless of hardware architecture, this provides an advantage in the *Consistency* dimension, even to experienced users switching from one processor to another.

The graphical icons employed in the visual syntax also offers an advantage in the *Role Expressiveness* dimension: although the meaning of mnemonics such as `add` or `mov` are straightforward, many instructions are much more obscure, such as `lea` (Load Effective Address) or `cvtdq2pd` (Convert Packed Doubleword Integer to Packed Double-Precision Floating Point Value). Although this is furthered by the colours of tiles used in the high-level visual syntax, such as green for instructions, yellow for invocation and blue for looping and iterative structures, the same can be said of any modern text editor with syntax highlighting. For this reason, *Enthusi*ASM offers no real advantage in the *Secondary Information* dimension: although comments can be added for each instruction and procedure, the same is true of textual assembly.

The high-level macros in *Enthusi*ASM also offer a detachment from the hardware. If we consider the problem domain of general-purpose assembly to be the hardware architecture itself, this provides an advantage in the *Closeness of Mapping* dimension. Although these match with the macros in textual assembly compilers, the ability to create new procedures by specifying the procedure name for an invocation tile manually in text gives the user the freedom of top-down design, allowing *Enthusi*ASM to be considered abstraction-tolerant in the *Abstraction Gradient* dimension, whereas textual assembly languages are by nature abstraction-hating [4]. The continuous execution environment for debugging sessions provided by the simulation layer, in which the user is free to add and remove tiles during step-wise execution of a procedure, allows a significant advantage in the *Progressive Evaluation* dimension in a similar manner to Prograph [4, 6].

Many VPLs are considered to have disadvantages in the *Viscosity* and *Premature Commitment* dimensions [4], however, *Enthusi*ASM does not lock the user into any inflexible structures such as LabVIEWs iterative structures [4, 20], and even high-level tiles may be inserted or removed at will, with *Enthusi*ASM automatically providing the indentation for nested structures. Procedures can be moved from one assembly file to another in the **Navigator** window, with *Enthusi*ASM resolving dependencies such as variable and constant allocations. *Enthusi*ASM therefore has an advantage over most VPLs in these dimensions, and may even have a small advantage over textual assembly.

This ability to resolve dependencies in the procedures also gives some flexibility in the *Hidden Dependencies* dimension, at least for procedures local to a particular project. The **Navigator** window explicitly shows these dependencies through the use of solid arrows detailing the calling hierarchy. Since the variables and constants are defined entirely within the **Memory** window, changing variable names here immediately changes variable names in tiles that examine or change these memory locations throughout the procedures within the same assembly file: this is roughly equivalent to a search-and-replace [4] in textual assembly although it is entirely automated by the GUI. Where global dependencies are concerned, however, such as when a global macro available to all projects is changed, projects in *Enthusi*ASM are not made aware of this until compilation, just as with textual assembly.

There are advantages and disadvantages in the *Visibility* dimension. Just as with textual assembly, given

unlimited screen dimensions, the entire code sequence for any procedure is visible, and any two procedures may be juxtaposed for comparison by opening two **Tiles** windows. The hierarchical organization and search capabilities provided by the **Available Tiles**, **Processor** and **Memory** windows help the user to quickly locate an instruction, register, variable or constant. However, when using high-level assembly, the input and output stacks of invocation tiles are only visible upon request: this is a necessary tradeoff in order to maintain a consistent tile height to aid in program comprehension.

Finally, since *Enthusi*ASMs visual tiles map one-to-one with textual assemblys instruction mnemonics and high-level structures, allowing the visual syntax to be just as terse, there is no advantage or disadvantage in the *Diffuseness* dimension. In fact, we see in Plate 1 an example where the visual and textual version of the same procedure is shown in roughly the same amount of screen real estate, noting that the comments are not shown in this figure due to space considerations. This one-to-one mapping also means that there can be no advantage in the *Hard Mental Operations* dimension, although solutions to the problem at hand may be somewhat easier to visualize during development since the logical thought process of which instructions must precede and follow other instructions is the same in both visual and textual assembly.

4. Conclusion

We have introduced a tile-based VPL for the development of assembly language programs, targeted towards students of machine organization and computer languages, and programmers of higher-level languages who program in assembly infrequently but wish to include high-performance assembly modules in their applications. We have illustrated the use of this language with a realistic prototype IDE, and have formally compared the system to textual assembly languages.

Other than full implementation, future work may include porting the proposed system to other hardware architectures, assembly compilers and operating systems, such as the new Intel 64-bit Itanium architecture [10] with the new generation of 64-bit Linux operating systems. Since the *Enthusi*ASM IDE would be developed with a highly portable Java GUI, porting to another operating system such as Linux would involve simply redefining the list of available kernel-specific macro tiles. Porting to another processor, however, would involve defining the available instruction set, registers and addressing scheme, the syntax of compiler directives, creating a simulation for each instruction and register for debugging and reverse execution, and the inclusion of any arbitrary restrictions for the particular architecture to check instruction legality.

References

[1] M. Beaumont, D. Jackson. Visualisation as an aid to low-level programming. *Proc. IEEE Frontiers in Education Conf. 27*, 3:1158–63, 1997.

[2] P.A. Carter. *PC Assembly Language*. pp.4–43, 2005, [http://www.drpaulcarter.com/pcasm].

[3] W.F. Decker. Making concepts and phenomena visual in machine and assembly language programming. *Proc. SIGCSE Technical Symposium on Computer Science Education 18*, pp. 432–41, 1987.

[4] T.R.G. Green, M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing,* 7(2):131–74, 1996.

[5] T. Grysztar. *Flat Assembler (FASM)*. 2004–2005, [http://flatassembler.net].

[6] D. Hils. Visual Languages and Computing Survey: Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing,* 3(1):69–101, 1992.

[7] R. Hyde. *The Art of Assembly Language*. 2001.

[8] R. Hyde. *The Great Debate: A series of essays on the need for assembly language in modern systems*. 2000, [http://webster.cs.ucr.edu/Articles/GreatDebate].

[9] Iczelion. *Win32 Assembly Tutorials*. [http://win32assembly.online.fr].

[10] Intel Corp. *IA-32 Intel®Architecture Software Developer's Manual*. ON: 253665, 253666, 253667 and 253668, 2005.

[11] Intel Corp. *Intel Architecture Optimization Manual*. ON: 242816-003, 1997.

[12] MASM32 Development Team. *MASM32 Project*. 1998-2004, [http://www.masm32.com].

[13] Microsoft Corp. *Microsoft Macro Assembler (MASM)* and *Microsoft Windows®*. 1981–2005.

[14] M. Newsome, C.M. Pancake, C. Ward. Visual execution of assembly language programs. *Proc. ACM Conf. on Computer Science*, pp. 38–43, 1993.

[15] A. Repenning, A. Ioannidou, J. Zola. AgentSheets: End-User Programmable Simulations. *Journal of Artificial Societies and Social Simulation*, 3(3), 2000.

[16] T. Shanley. *Pentium®Pro Processor System Architecture*. MindShare, Addison–Wesley, pp. 61–80, 1996.

[17] J. Sharp. *Data Flow Computing*. Ellis Horwood, pp. 17-38, 1985.

[18] Z. Shi, P.T. Cox. EasyAssembly: A Real Visual Assembly Language. *Proc. IEEE Newfoundland Electrical and Computer Engineering Conf. 11*, 2001.

[19] S. Tatham, J. Hall, H.P. Anvin, J. Fine, K. Bennett, G. Clark, A. Crabtree. *NetWide Assembler (NASM)*. 2003, [http://nasm.sourceforge.net].

[20] K.N. Whitley. Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages and Computing*, 8(1):109–42, 1997.