

SIMx86: An Extensible Simulator for the Intel*80x86 Processor Family

Alan R. Shealy
Dept. of Computer Science

Abstract

The complexity of modern processors requires that their development be paralleled by the development of a simulator to guide design decisions. In addition to facilitating the development of the processor, a simulator can also facilitate the development of compiler techniques to exploit the increased speed and functionality of the new processor. Finally, since successful processors are succeeded by newer and better processors, an established simulator can be invaluable in aiding the development of the successor processor. However, the advantages that a simulator provides to a developing architecture require that the simulator be easy to extend and modify.

In this paper, we exploit object technology to present the design and implementation of SIMx86, an execution-driven simulator for the 80x86 processor family. We begin by describing the design of a domain model for processor simulators. We demonstrate the extensibility of our design by extending it to include first the Intel 8088 processor and then the 8086 processor. We further demonstrate extensibility by incorporating debugging facilities into our simulator model. To evaluate the performance of our SIMx86 simulator, we compare execution times with an existing simulator, the SimpleScalar *sim-fast* simulator. Our experiments indicate that SIMx86 is competitive with the *sim-fast* simulator. In most cases, the *sim-fast* simulator was about three times faster than SIMx86. However, the ease of modification, extension and maintenance provided by our design offsets the performance gains provided by the traditional approach to simulator construction.

keywords: Methodology, object-oriented, execution-driven simulation, Intel 80x86

* Intel, Pentium, and PentiumPro are registered trademarks of the Intel Corporation.

1 Introduction

The complexity of modern processors requires that their development be paralleled by the development of a simulator to guide design decisions. In addition to facilitating the development of the processor, a simulator can also facilitate the development of compiler techniques to exploit the increased speed and functionality of the new processor. Finally, since successful processors are succeeded by newer and better processors, an established simulator can be invaluable in aiding the development of the successor processor. However, the advantages that a simulator provides to a developing architecture require that the simulator be easy to extend and modify.

The major obstacle in using simulation to facilitate development of new architectures is that traditional simulation approaches are not flexible enough to permit easy extension or modification of the underlying simulation model. Although most simulators are parameterized to provide flexibility[4, 18, 22], parameterization alone is not adequate to permit the modification and extension required for reusing an existing simulator to facilitate the development of a new architecture. Thus, researchers interested in developing new architectures must begin with the daunting task of constructing a new simulator.

In this paper, we exploit object technology to present the design and implementation of SIMx86, an execution-driven simulator for the 80x86 processor family. Object technology enables straightforward construction of a simulator for a processor by permitting each component of the processor to be represented directly in the simulator by an object. The relationships between these objects in the software reflect the logical relationships between processor components. We begin by describing the design of a domain model for processors and the construction of simulators for processors. We then demonstrate the extensibility of this model through a series of extensions. We first extend the domain model to include a simulator of the Intel 8088 processor. We then extend the design further to include the 8086 processor. In the final extension to the design we incorporate debugging facilities into our simulator model. The mechanisms that we use to provide extensibility are inheritance and genericity (templates in $C++$).

Since object technology can be inefficient due to dynamic binding, we exploit object technology further to attain faster execution in our $C++$ implementation. Traditional approaches to increasing execution speed have focused on (1) pre-decoding instructions to an intermediate representation [3, 12, 24, 27], and (2) cross

compilation [5, 6, 7, 10, 20, 27]. In our approach, each instruction in memory is decoded only once and represented by an object whose state directly reflects the execution of that object. This process of pre-decoding instructions results in increased performance, since the decode step need not be repeated during execution.

To evaluate the performance of our SIMx86 simulator, we compare execution times with an existing simulator, the *SimpleScalar* simulator[4]. The SimpleScalar system includes four simulators ranging from the fastest and least detailed simulator, *sim-fast*, to the most complicated and detailed simulator, *sim-outorder*. All SimpleScalar simulators are coded in C[15]. For our experiments, we chose the fastest and least detailed SimpleScalar simulator: *sim-fast*. *Sim-fast* is written in less than 300 lines of code, does no time accounting, executes each instruction serially and assumes no cache[4]. We chose *sim-fast* because it represents an exceptionally efficient baseline simulator that is likely to produce excellent execution times. Our experiments indicate that SIMx86 is competitive with the *sim-fast* simulator. In most cases, the *sim-fast* simulator was about three times faster than SIMx86. However, the ease of modification, extension and maintenance provided by our design offsets the performance gains provided by the traditional approach to simulator construction.

In the next section, we provide background about the 80x86 architecture and about simulation techniques. In Section 3 we describe our model and in Section 4 we describe extensions to the model. In Section 5 we present the results of our experiments, in Section 6 we describe other relevant simulators and we draw conclusions in Section 7.

2 Background

In this section, we provide background about 80x86 processors and approaches to simulating computer architectures. We present the Intel 80x86 family of processors with a short description of their evolution, highlighting the innovations in each new processor. We then examine simulation techniques that have proven successful for simulating processor execution.

2.1 The 80x86 Processor Family

In 1978, Intel introduced the 8086 microprocessor, an extended accumulator architecture. The 8086 architecture provided an assortment of 16 bit registers, many were special purpose, and twenty bit, little-endian physical addresses to access memory. Instructions provided three operand types: *memory*, *register*, and *immediate*. Instructions could combine these operand types in any manner, excluding two memory operands. The 8086 architecture included a wide assortment of memory addressing modes[13, 19, 28].

The Intel architecture became established in the personal computer arena in 1980, when IBM chose a version of the 8086 as the processor for its PC's. The 8086 uses a 16 bit external bus to connect the processor to main memory. However, to keep the cost of PCs low, IBM chose to use the 8088, a version of the processor with an 8 bit bus [13].

Subsequent processors developed by Intel extended the functionality of the 8088/86. All successive processors were backward compatible with previous processors in the family; however, each successor processor used new technology to increase efficiency and processor speed over previous processors.

The 80286 was the successor to the 8088/86. The major contribution of the 80286 was to increase the physical address size to 24 bits by segmenting the memory. The 80386 added memory paging while increasing addresses to 32 bits. In addition, data registers were increased to 32 bits.

No substantial changes to memory addressing and registers have been made in processors that followed the 386. Rather, subsequent 80x86 processors have concentrated on fine tuning the micro-architecture of the processor to increase performance. The 80486 was the first pipelined processor in the family. The Pentium processor subsequently utilized two concurrent 486 pipelines to achieve two-way superscalar performance. The Pentium Pro, the successor to the Pentium, exploits branch prediction, data flow analysis, and speculative execution (collectively termed "Dynamic Execution" by Intel) to further increase execution performance. The out-of-order execution of instructions by the Pentium Pro results in three-way superscalar performance [8].

The evolution of the 80x86 processor family illustrates the fact that simulators for these architectures must be flexible enough to permit easy extension and modification. An extensible simulator design allows for code reuse and minimizes effort when developing a simulator for the next processor in the series.

2.2 Approaches to Processor Simulation

Traditional computer architecture simulations made use of *trace-driven* simulation. Trace-driven simulation has been used effectively to evaluate cache performance [23]. A trace of the instructions executed by the processor is recorded in a file and then later interpreted by the simulator. Often trace files are large and difficult to obtain [2].

Execution-driven simulation provides an alternative to trace-driven simulation. Instead of interpreting the instructions in a trace file, the simulator actually interprets instructions in a binary executable [18, 22]. Since cross-compilers are often readily available, executables are much easier to obtain and store. In addition, execution-driven simulators can exploit the structure of the program under execution by pre-decoding instructions. Optimizations based on the pre-decoded instructions can produce large performance gains for programs that contain loops.

3 An OO Design for Simulators

Object technology provides a straightforward way to build a simulator for a processor by allowing each component of the processor to be represented directly in the simulator by an object. The relationships between these objects in the software reflect the logical relationships between processor components. Our object model for the design of processor simulators is shown in Figure 1. This model is a domain model that can be extended to various target architectures. In the object model, the Class `ProcessorSimulator` represents a simulator itself that comprises instances of the various components of a processor. That class is instantiated and used to run a simulation. The constructor for the class accepts a parameter specifying a filename for the binary executable to be simulated. Method `runSim()` is invoked to perform a simulation run.

The kinds of components that comprise a processor—the registers, bus interface unit, bus, and memory—are represented by the classes in the model. The relationships between these components are also reflected by the model; the association between a CPU and a bus interface unit (BIU) illustrate their relationship. A single CPU is represented in the simulator as an instance of Class `CPU`. The system bus will be modeled as an instance of Class `Bus`, connected to the Class `CPU` instance by an instance of Class `BIU`. An instance

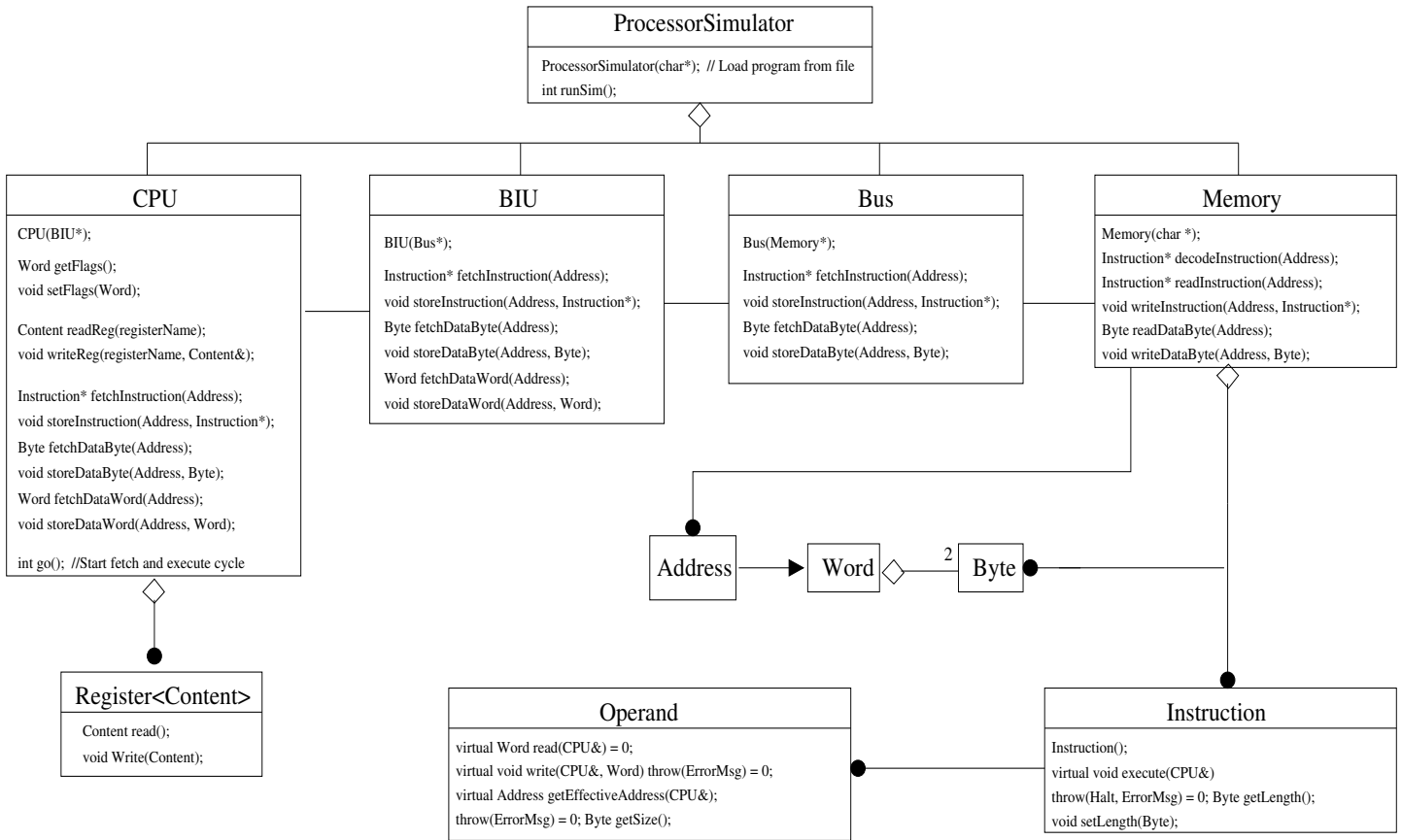


Figure 1: The object model for the processor simulator domain.

of Class `Register<Content>` is created for each register in the processor. The angle brackets in this class indicate that Class `Register` is a *generic*— or *template*—class, parameterized, in this case, by the type of its contents. For example, parameters to the `register` template might be Class `Word`, Class `Address`, or Class `Byte`. Using a generic `Register` class leads to extensibility, since registers of different sizes can be instantiated without the need for defining new classes.

Class `Memory` represents an association between addresses and storage locations (bytes or words). In addition, Class `Memory` is an aggregation of bytes and instructions, where an instruction corresponds to a machine instruction stored in memory. Our motivation for viewing memory this way is two-fold:

- *Faster simulation of executing instructions.* By representing each instruction in the code, we have in effect pre-decoded each instruction, saving that time in the simulation.
- *Extensibility of the instruction set.* More instructions can be added to the simulator by creating new subclasses of Class `Instruction` and implementing a single method, `execute()` (see Section 4).

Although representing each instruction in memory as an object results in a cost of more memory, the benefits

of extensibility and good performance outweigh this penalty. The decision limits simulation to execution of programs for which the instructions in the code segments do not change once they are loaded. This means we can support simulation of code produced by most compilers for high-level languages. However, it precludes some optimized assembly language programs that rely on self-modifying code to increase execution speed.¹

The `Instruction` class is also associated with Class `Operand`. Each instruction can take zero or more operands which are each represented by an object. Again, we have chosen extensibility at a cost of memory with the `Operand` class. By creating subclasses of `Operand` which define the `read()` and `write()` Class methods, we can easily add new operand addressing modes to the instruction set.

Some components of modern processors, such as cache memories and instruction pipelines, are missing from this model. By adding classes for these components to Figure 1, we can further extend the design model to create a framework for processors and their simulators.

4 Extensibility of the Design

Our work in processor simulation was originally motivated by our interest in what would comprise an object-oriented design for a simulator and by an observed need for an extensible, modifiable simulator for a Pentium Pro processor. Our first challenge was to find a design. We decided to focus on simpler processor architectures in the Intel family and work our way “up” to the Pentium Pro. The goal for our software architecture was extensibility—that is, an architecture that would permit our simulator code to easily adapt to new features of processor architectures or even features of the simulator itself. At the same time, the simulator needed to be efficient. Early experiments with a prototype of our object-oriented design indicated that execution speeds can be faster than a system built with a traditional design, but at a cost of using more memory[25]. Of course, efficiency and extensibility are two competing goals. When compromises are required, we have consistently chosen extensibility as our primary consideration. In our approach, extensibility is based on the modification (reuse) of existing code, primarily through the mechanisms of inheritance and generics(templates in C++) provided by object-oriented programming languages. That is, we extend the capabilities of existing simulator code by incrementally reusing and adding to the simulator’s code base. This approach is different from an approach in which extensibility is achieved via simulator parameterization[4, 18, 22]. We believe

¹At some point we may support self-modifying code, but it is not a priority.

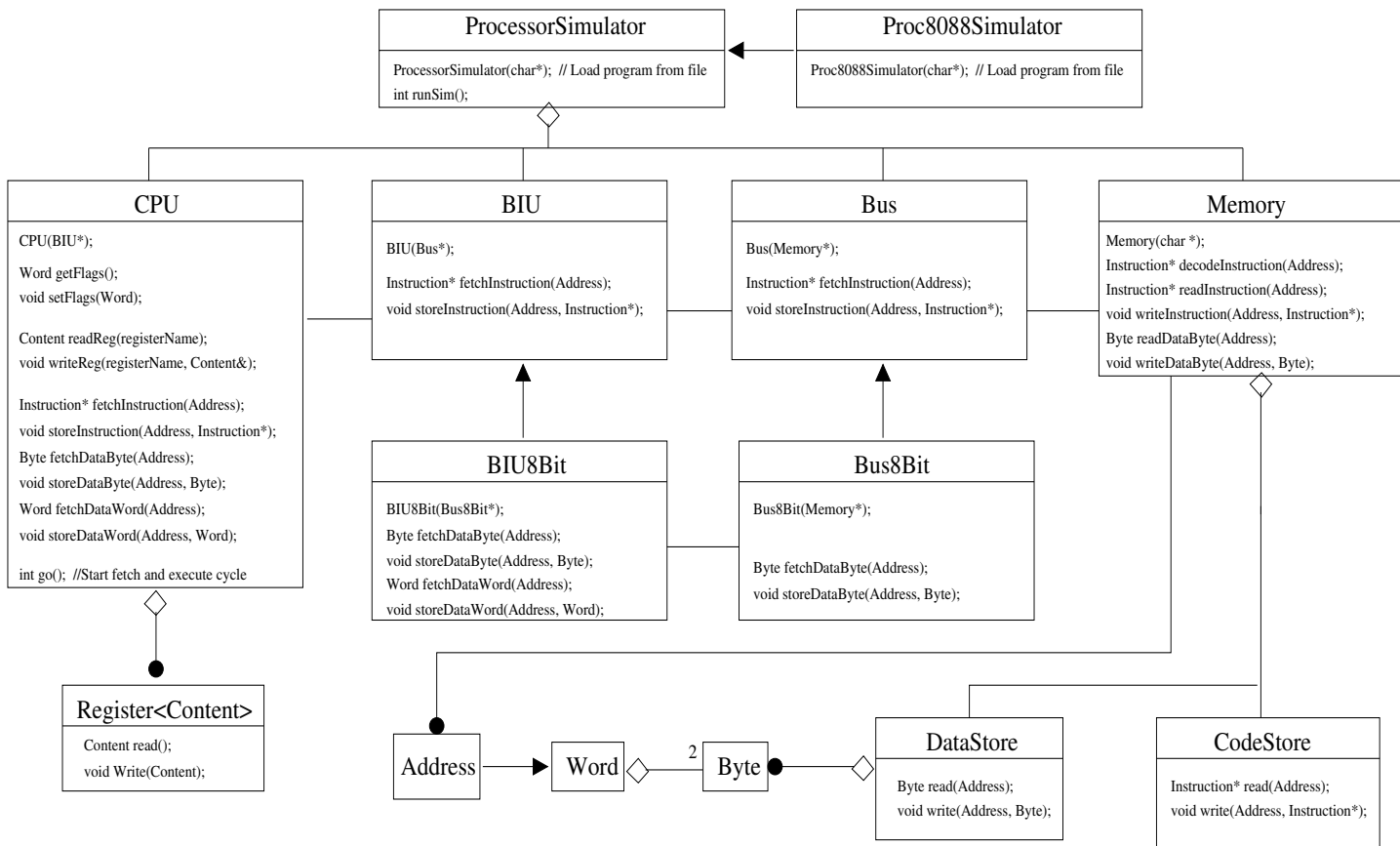


Figure 2: The object model extended for the Intel 8088 simulator.

our approach is more flexible and maintainable. Our goals for extensibility include an ability to add new instructions, to change the way instructions are executed, to introduce instruction pipelines and micro-operations, to experiment with different memory hierarchies (levels of caches), and to change the statistics collected by the simulator.

We now describe three ways that we have successfully, and easily, extended our domain model to create simulators for the Intel 80x86 processor family. Each extension was implemented using *C++*. Collectively, we refer to the 80x86 simulation framework as *SIMx86*.

4.1 An Intel 8088 Simulator

The first extension to the object model was made in order to build a simulator for the Intel 8088 processor. Figure 2 shows the extensions that were made. Our extensions to the model are made primarily using the object-oriented technique of inheritance. The *Proc8088Simulator* Class is a subclass of Class

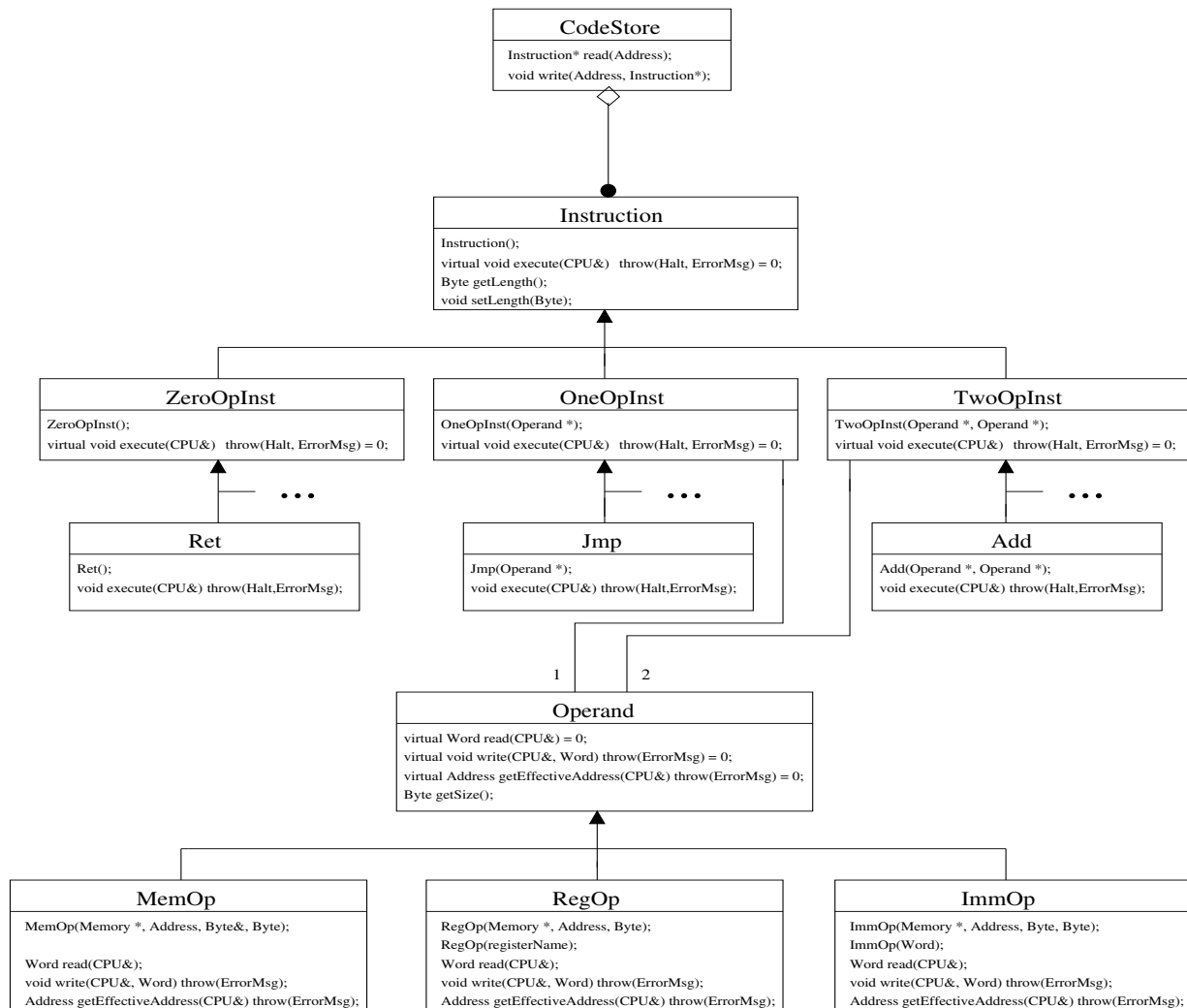


Figure 3: The Code Store model for the Intel 8088 simulator.

ProcessorSimulator. It aggregates components specific to the 8088 processor, such as an eight bit bus interface unit and eight bit data bus. Inheritance is used to create classes BIU8Bit and Bus8Bit as well. These classes may be used in simulators for any architecture that makes use of an eight bit data bus.

The model reflects an important design decision that we have made for our simulators. We have separated the portions of memory that store instructions from the portions that store data. Although Class Memory still represents an association between addresses and storage locations (bytes or words), we have further decomposed it into instruction and data stores. Data store is an aggregation of bytes and code store is an aggregation of instructions. This decision was made in an effort to make the organization of the Class Memory more straightforward and maintainable. Figure 3 displays the 8088 simulator's code store in further

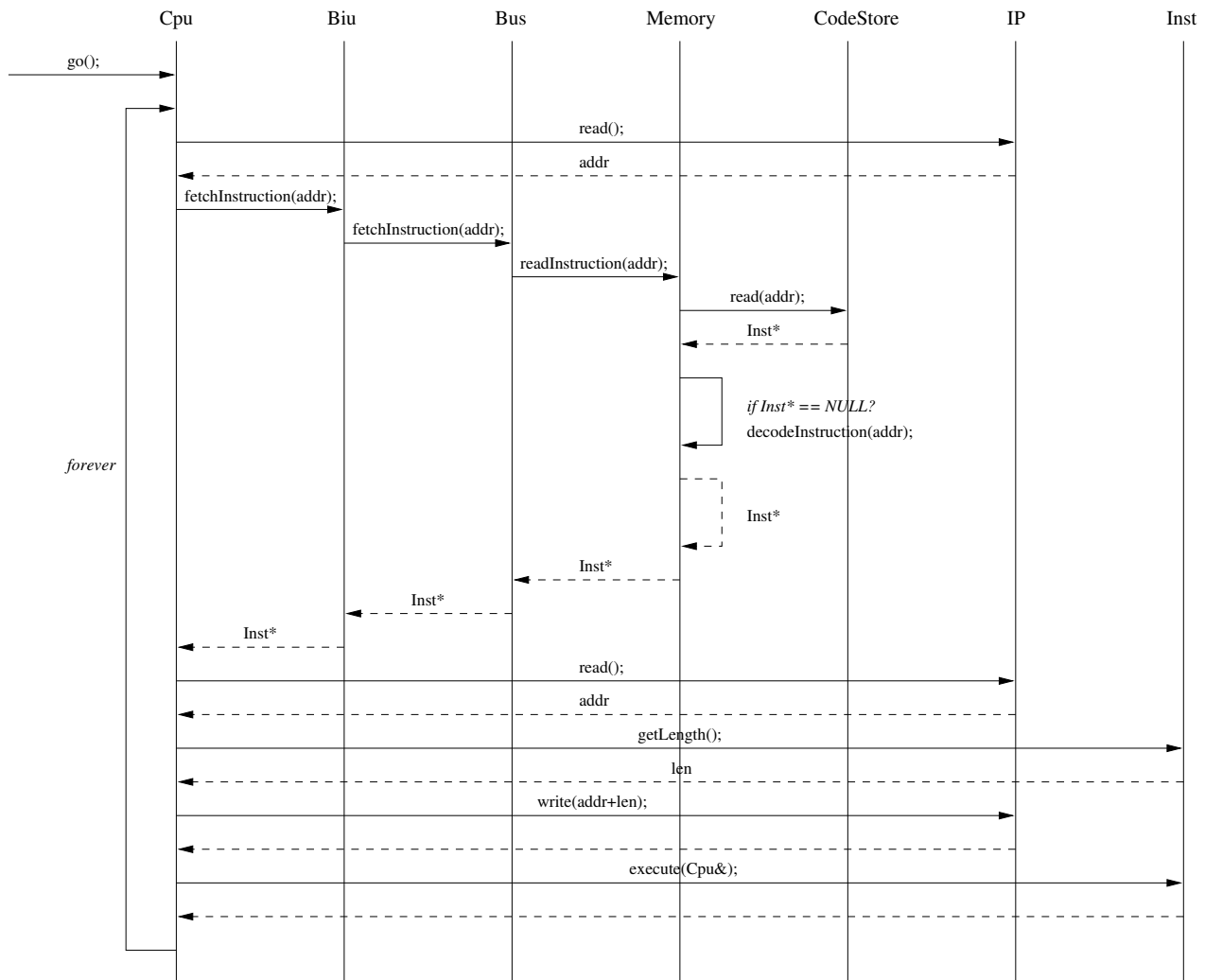


Figure 4: Object Interaction Diagram for the simulator's fetch and execute cycle.

detail. Subclasses of Class `Instruction` and Class `Operand` have been created which are specific to the Intel 8088 instruction set. Three subclasses of `Instruction` have been created corresponding to instructions with zero, one, and two operands. Each of these classes, in turn, is a base class for many concrete instruction classes such as `Ret`, `Jump`, and `Add`. Also, three subclasses of Class `Operand` are created which correspond to the operand types present in 8088 instructions: memory, register, and immediate.

Figures 4 and 5 show representative interactions between the objects that exist in the simulator. Figure 4 shows the objects and their interactions with respect to the fetch-execute cycle of an 8088. In an interaction diagram, the names across the top represent objects—that is, instances of classes in the object model—that are involved in implementing some aspect of processing, in this case the fetch-execute cycle. An arrow

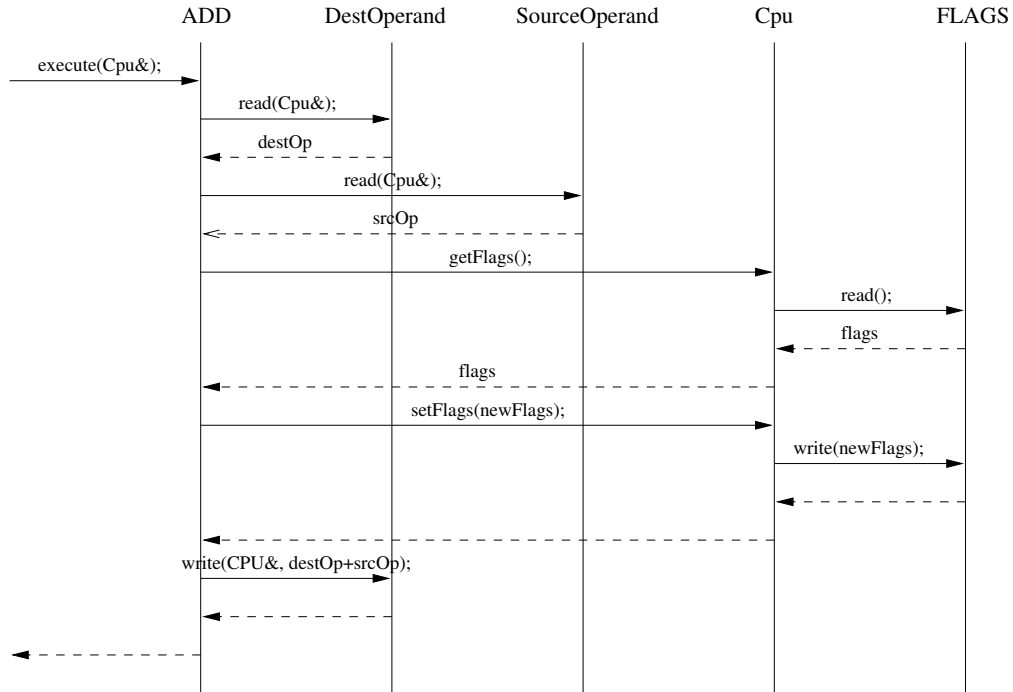


Figure 5: Object Interaction Diagram for the execution of a ADD instruction.

represents a method invocation (member function call in C++). A dashed arrow represents a return from a method invocation. Time increases from the top of the diagram to the bottom. The fetch-execute cycle shown for the 8088 is representative of simulators for various processors built from these classes. Note that an instruction is executed by invoking its `execute()` method. This method invocation shows a tradeoff that we have made in our design for the simulator. Since the concrete instruction class (`Add`, `Jump`, etc.) is not known until runtime, the `execute()` message must be dynamically bound with a runtime dispatch. Thus, a side effect of using inheritance to increase extensibility in the instruction set is somewhat degraded performance.

In order to combat this performance loss due to runtime dispatching, we pre-decode the program instructions. Traditional simulators fetch the instruction from memory, decode it, then execute it. In our object-oriented design, the instruction is only decoded the first time it is executed. Experiments with profiling measurements for `SIMx86` indicate that decoding each instruction once, can result in tremendous performance increases. On the other hand, decoding the instructions at each encounter can result in a 50% increase in execution time for an $O(n)$ program that required one second to simulate. Larger and more

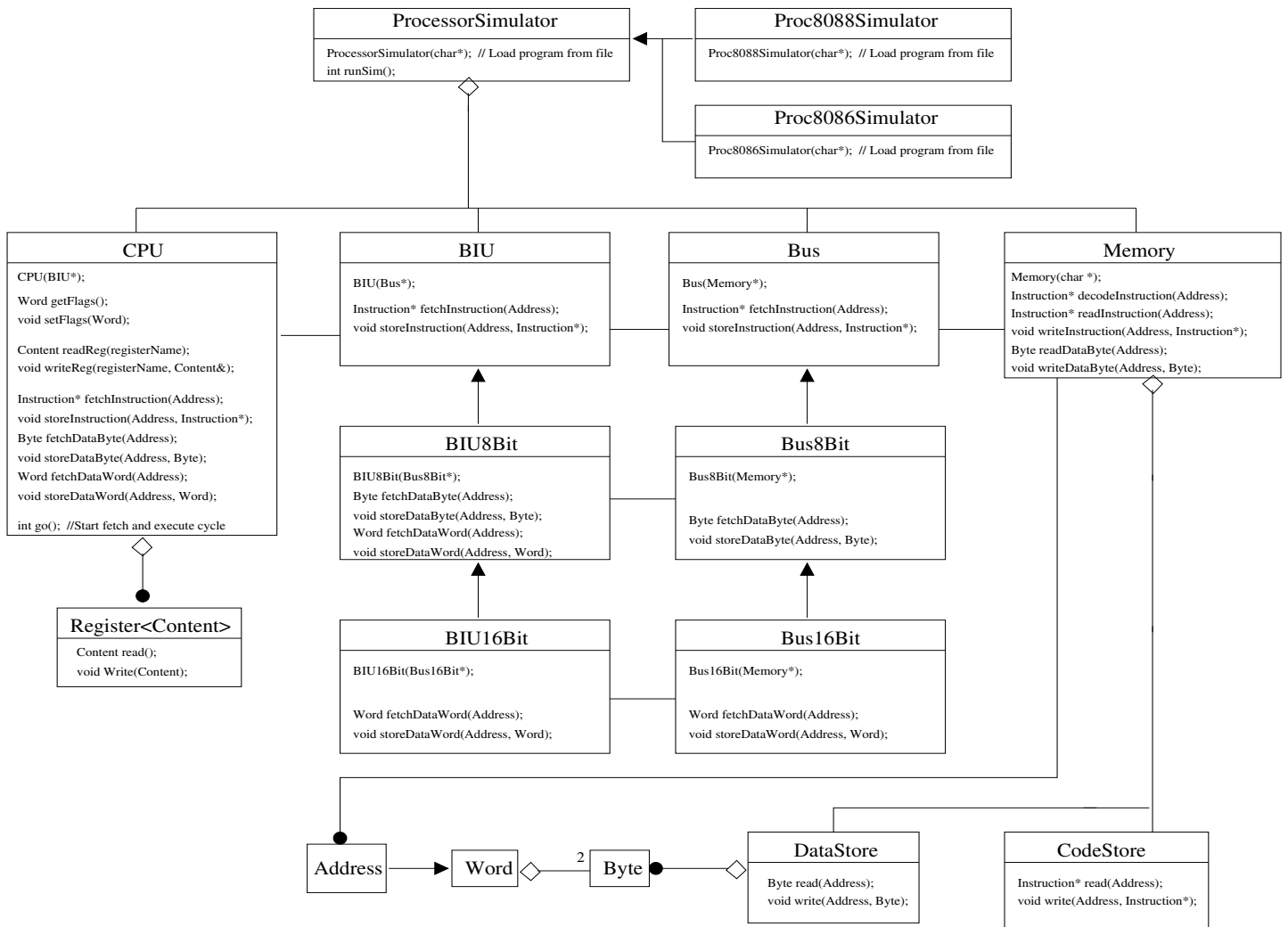


Figure 6: The simulator object model modified for the Intel 8086 processor.

complex programs suffered even greater performance penalties.

Figure 5 models the actions of an Add instruction; other instructions are modeled similarly. Note here as well the tradeoff between performance and extensibility. When `read()` and `write()` messages are sent to instruction operands, they must be runtime dispatched.

4.2 Extending to the 8086

We extended the code for the 8088 simulator we developed in C^{++} to simulate the Intel 8086 processor. Let us make clear that the second simulator does not embed simulators for both the 8086 *and* the 8088. There are two simulators, one for each, but they are constructed largely from the same code. Thus, we have created

a new subclass of Class `ProcessorSimulator` called `Proc8086Simulator`. Figure 6 shows the extensions to the object model that are necessary for the 8086 simulator.

The difference between the Intel 8088 and the 8086 processors is the latter's incorporation of a 16-bit data bus to enable word-access of memory. The simulation of an extended data bus is straightforward to implement in our design by simply using the object-oriented technique of inheritance. Subclasses of Class `BIU8Bit` and Class `Bus8Bit` are created called Class `BIU16Bit` and Class `Bus16Bit`, respectively. This represents another decision that we have made in the design of our simulators. From a modeling perspective, a clearer approach might be deriving Class `BIU16Bit` and Class `Bus16Bit` directly from Class `BIU` and Class `Bus`, respectively. This would create a flatter inheritance hierarchy. However, we have favored code reuse in our approach. Since the 16 bit bus must also have the capability of fetching and storing a single byte, it in effect augments the interface of an 8 bit bus.

Class `Bus16Bit` adds the Class `fetchDataWord(Address)` and Class `storeDataWord(Address, Word)` methods to the Class `Bus8Bit` interface. The corresponding methods of the Class `BIU16Bit` class are overridden to use the new methods in Class `Bus16Bit` instead of, for example, using two invocations of the Class `fetchDataByte(Address)` method. No modification or extension to the instruction and operand classes is necessary since the 8088 and 8086 use the same instruction set. We are continuing to extend the model for the 80286 and also adding caching and pipelining components to the model.

4.3 A Simulator with Debugging Support

As an experiment, to see the impact of adding additional features to the simulators, we implemented some primitive debugging capabilities within the simulator—for example, to single-step instruction execution and to display the contents of registers. Again, inheritance was used to create new classes which would enable program debugging. Figure 7 displays the additions to the object diagram necessary for the debugging simulator. Class `DebuggingSimulator` is a subclass of Class `ProcessorSimulator` and adds the Class `debug()` method to its interface. Class `DebuggingCPU` is one of the components of Class `DebuggingSimulator`. The `DebuggingCPU` class is a subclass of Class `CPU` and adds the Class methods `trace()`, `printRegs()`, `dumpMemory()`, and `help()` to its interface.

In addition, modification was made to the instruction and operand classes for the 8088 instruction set.

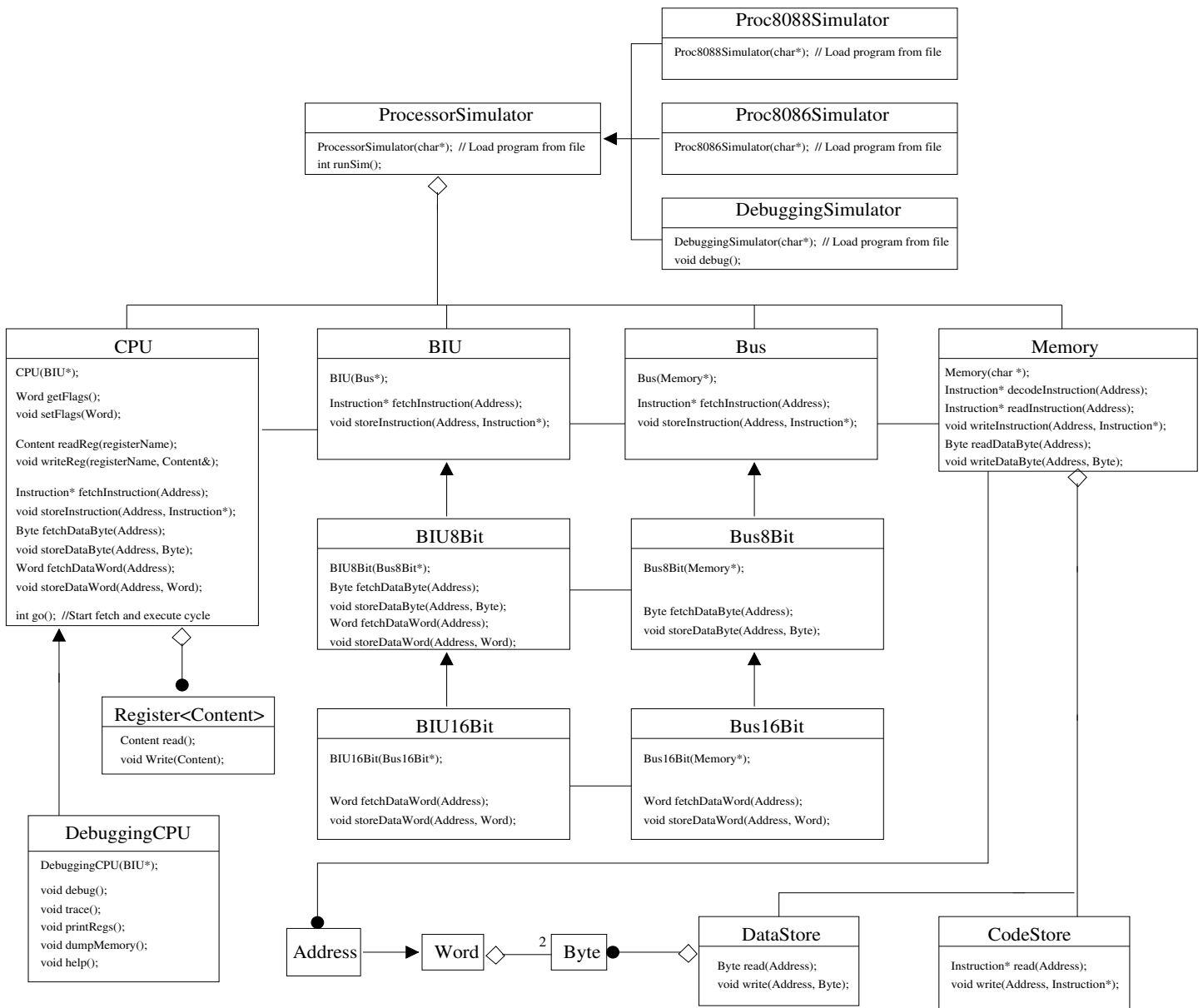


Figure 7: Augmented object diagram for a simulator with a debugger interface.

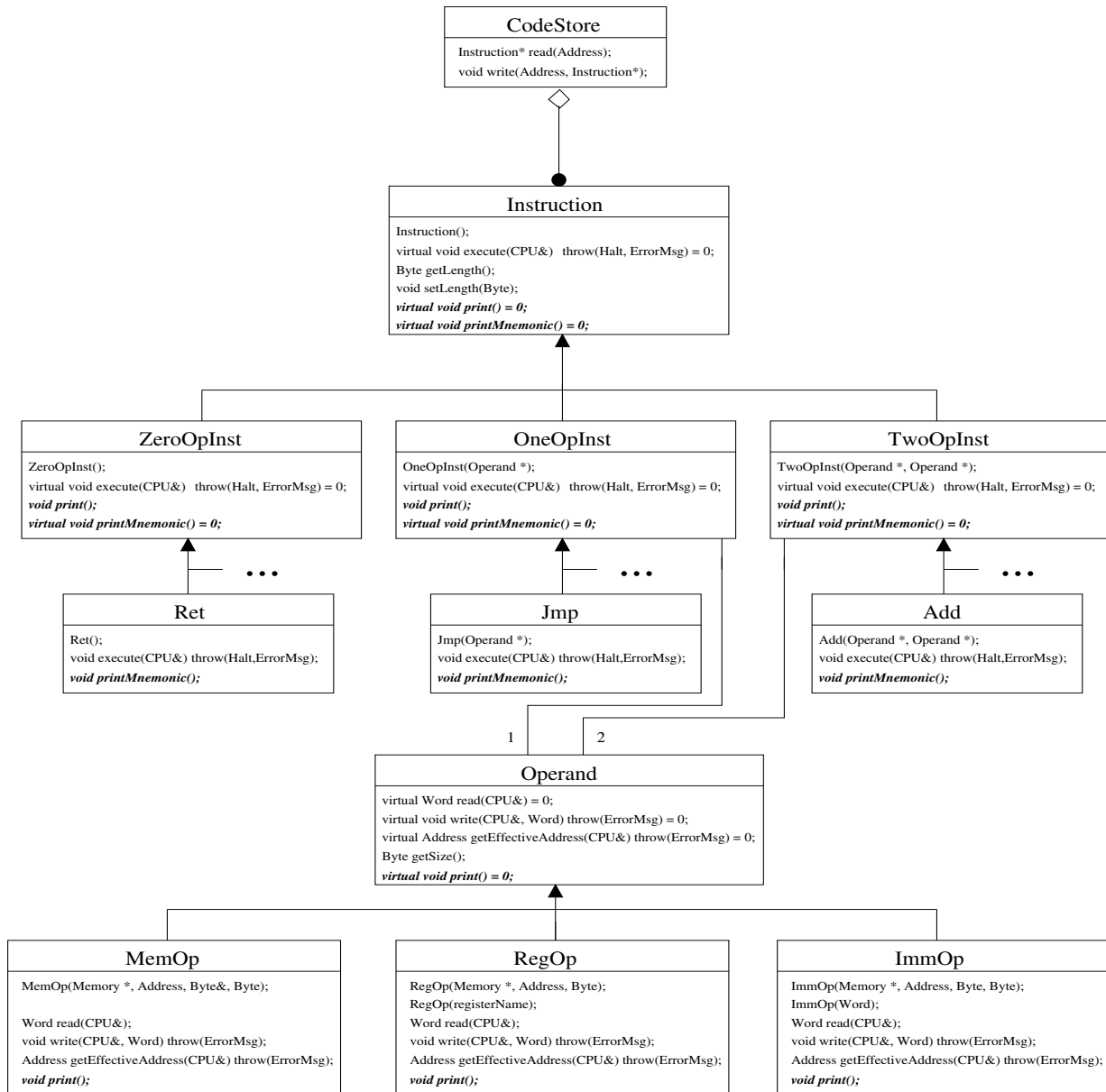


Figure 8: Modifications to the instruction and operand classes necessary for debugging support.

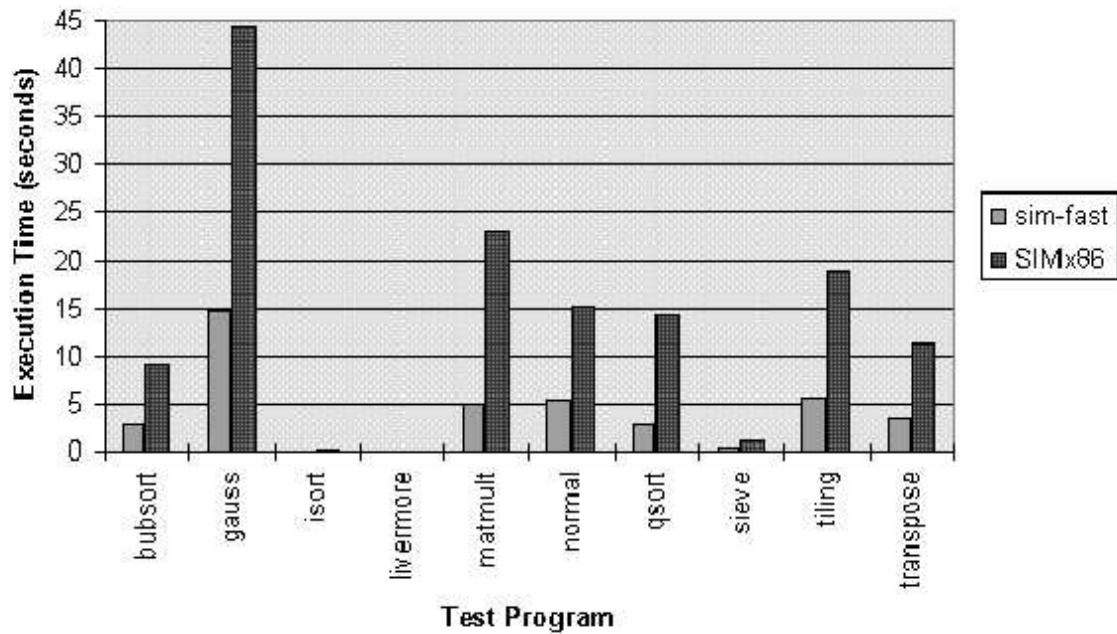
These changes are shown in Figure 8. A set of `Class print()` methods were added to the subclasses of `Class Instruction` and `Class Operand` to support the printing of instructions in a symbolic form. An alternative to these `print()` methods might be overloading the `cout` operator in a `C++` implementation.

In lieu of a `Class trace()` operation within the interface of `Class DebuggingCPU`, we are investigating the definitions of *states* of a CPU with respect to a simulation. Among those states will be a *single-stepping state*, which will allow single stepping.

5 Performance

In Sections 3 and 4 we presented the `SIMx86` design for 80x86 simulators that exploits object technology to produce a design that is easy to extend, easy to modify and easy to maintain. When making design decisions, we have always favored extensibility and easy maintenance, over efficiency. For example, we define the operations in class `Instruction` polymorphically, when a templated class `Instruction` is likely to be more efficient. In this section, we present some performance results to determine the efficiency of our simulator. We have chosen ten test programs and we compare simulation times for these programs using our `SIMx86` simulator for the Intel 8086 and the *SimpleScalar* simulator for a MIPS-like processor[4]. The *SimpleScalar* system includes four simulators ranging from the fastest and least detailed simulator, *sim-fast*, to the most complicated and detailed simulator, *sim-outorder*. All *SimpleScalar* simulators are coded in C[15]. For our experiments, we chose the fastest and least detailed *SimpleScalar* simulator: *sim-fast*. *Sim-fast* is written in less than 300 lines of code, does no time accounting, executes each instruction serially and assumes no cache[4]. The *sim-fast* simulator separates the code segment from the data segment and uses fast macro substitution for pre-decoded instructions. We chose *sim-fast* because it represents an exceptionally efficient baseline simulator that is likely to produce excellent execution times.

Our testsuite is composed of ten C programs that include both scientific and general purpose workloads. However, inputs to the two simulators, `SIMx86` and *sim-fast*, are quite different; although both simulators take binary executables, the form of the executables is different. To create 8086 binary executables for `SIMx86`, we use a C compiler for the PC to produce 8086 assembly code. The 8086 assembly code is assembled using Wolfware Assembler, or `WASM`[26], to create an executable COM file. A COM file is a command processing file that contains executable commands or statements. The code and data in COM files are not



Program	Data Set Size	<i>SIMx86</i> Instructions Simulated in Thousands	<i>sim-fast</i> Instructions Simulated in Thousands	<i>SIMx86</i> Sim Time in seconds	<i>sim-fast</i> Sim Time in seconds	Native Exec Time in seconds
1. bubsort	5000	2268.84	5483.96	9.06	2.73	0.03
2. gauss	100x100	9336.67	17360.30	44.40	14.75	0.21
3. isort	100	50.61	141.73	0.22	0.09	0.01
4. livermore	400	8.81	22.98	0.04	0.04	0.01
5. matmult	50x50	5290.79	9177.55	22.85	5.06	0.10
6. normal	50x50	3613.58	10128.16	15.20	5.44	0.09
7. qsort	10000	3609.66	5476.26	14.34	2.83	0.05
8. sieve	10000	224.96	859.82	0.99	0.46	0.01
9. tiling	50x50	4506.53	10120.53	18.91	5.53	0.08
10. transpose	50x50	2641.44	6178.82	11.28	3.56	0.05

Figure 9: Performance Results for the Test Suite of 10 programs.

placed in different segments; thus, SIMx86 must distinguish code from data during the simulation. SIMx86 interprets the normal binary instruction set. Executables for *sim-fast* are produced by a version of gcc that cross-compile to the SimpleScalar instruction set, which is unique to the SimpleScalar simulators. Code and data are placed in different segments by the gcc compiler so that *sim-fast* does not have to distinguish code from data during simulation.

All experiments with the test programs were conducted on a Gateway 2000 with a 133 MHz Pentium processor running the Solaris 2.5 operating system. The programs were executed ten times and the execution times reported in this section are averages over these ten executions.

Figure 9 presents, in both graphical and tabular form, the results of our experiments. The first column of

the table in Figure 9 lists our test programs, the second column lists the size of the data sets used by the test programs, the third and fourth columns list the number of instructions simulated by our SIMx86 simulator and *sim-fast* respectively for each of the test programs, columns five and six list simulation times for our SIMx86 simulator and the *sim-fast* simulator respectively, and the final column lists the actual execution time of the native code compiled for the Pentium-133 and executed on the host hardware (i.e., not simulated).

The test programs listed in column one of the table in Figure 9 include the bubble sort, `bubsort`; a program that uses Gaussian elimination without pivoting, `gauss`[29]; an insertion sort, `isort`; the first livermore loop, `livermore`[21]; matrix multiplication, `matmult`[29]; a program to transform a matrix into *Hermite normal form*, `normal`[29]; the quicksort program, `qsort`; the sieve of Erosthathenes, `sieve`; a program that uses *tiling* to optimize data cache references, `tiling`[16]; and a program to perform matrix transposition, `transpose`.

The graph in Figure 9 overviews our efficiency comparison of SIMx86 and the SimpleScalar *sim-fast* simulator by comparing the total simulation time for each test program. The graph shows that *sim-fast* simulates each program in less time than SIMx86. The table in Figure 9 provides details to facilitate interpretation of the experiments. For example, the `gauss` program appears in the second row of the table where the second column of data lists the data set size of 10,000 memory locations (100x100), the third column lists the number of instructions simulated by SIMx86 as 9336.67 thousand instructions, and the fourth column lists the number of instructions simulated by *sim-fast* as 17360.30 thousand instructions. Note that the number of instructions simulated is different for the two simulators since they simulate two different instruction sets. The fifth column lists the simulation time for SIMx86 as 44.4 seconds and the sixth column lists the simulation time for *sim-fast* as 14.75. The final column for the `gauss` program lists the time to execute the native code on the Pentium-133 host as 0.03 seconds.

Analyzing the data presented Figure 9 in an attempt to compare the two simulators is not straightforward since they simulate different architectures. They are both functional simulators with no time recording and no cache. However, there are several differences between the two simulators. Most notable are the differences in the instruction set. SIMx86 makes use of complex instructions, in contrast to the RISC instructions used by *sim-fast*. In addition, the simulators use different register, bus, and memory sizes.

<i>Program</i>	<i>SIMx86</i> <i>Thousands of Inst</i> <i>Simulated per sec</i>	<i>sim-fast</i> <i>Thousands of Inst</i> <i>Simulated per sec</i>	<i>ratio</i>
1. <code>bubsort</code>	250.6	2008.0	8.01
2. <code>gauss</code>	210.3	1176.9	5.59
3. <code>isort</code>	230.0	1574.8	6.85
4. <code>livermore</code>	220.3	574.5	2.61
5. <code>matmult</code>	231.5	1814.9	7.84
6. <code>normal</code>	237.7	1861.8	7.83
7. <code>qsort</code>	251.9	1938.0	7.69
8. <code>sieve</code>	227.2	1869.2	8.23
9. <code>tiling</code>	238.3	1830.1	7.68
10. <code>transpose</code>	234.2	1735.6	7.41
Average	233.2	1638.4	6.97

Table 1: This table illustrates the ratio of instructions simulated per second for the SIMx86 and the SimpleScalar *sim-fast* simulators.

<i>Program</i>	<i>SIMx86</i> <i>Simulation Time</i> <i>in seconds</i>	<i>sim-fast</i> <i>Simulation Time</i> <i>in seconds</i>	<i>ratio</i>
1. <code>bubsort</code>	9.06	2.73	3.32
2. <code>gauss</code>	44.40	14.75	3.01
3. <code>isort</code>	0.22	0.09	2.44
4. <code>livermore</code>	0.04	0.04	1.00
5. <code>matmult</code>	22.85	5.06	4.51
6. <code>normal</code>	15.20	5.44	2.79
7. <code>qsort</code>	14.34	2.83	5.06
8. <code>sieve</code>	0.99	0.46	2.15
9. <code>tiling</code>	18.91	5.53	3.42
10. <code>transpose</code>	11.28	3.56	3.17
Average			3.08

Table 2: This table illustrates the ratio by simulation times for the SIMx86 and the SimpleScalar *sim-fast* simulators.

To facilitate our efficiency comparison of the SIMx86 and the SimpleScalar *sim-fast* simulator, Table 1 lists the ratios of the number of instructions simulated per second for the two simulators for each of the test programs. The final row of Table 1 lists the average ratio and shows that, generally, the *sim-fast* simulator simulates seven times the number of instructions simulated by SIMx86 in one second. However, we cannot make a meaningful comparison based on just Table 1, since the simulators are simulating different instruction sets. RISC instructions are faster to simulate than their complex counterparts, just as they are faster to execute in hardware. Thus, *sim-fast* might have an advantage in this comparison.

Table 2 provides another way to compare the simulators' performance. In this table, we simply compare the total simulation time for each test program. The final row of Table 2 shows that, generally, the total simulation time for a C program is three times greater for SIMx86 than for *sim-fast*. The comparison of total simulation time for each program is still influenced by the differences in architecture. However, the RISC versus CISC variable is somewhat constrained.

It is clear from each of the previous comparisons that *sim-fast* is faster than SIMx86. However, the preceding analyses give no indication of the relative simulation penalty incurred by both simulators. Perhaps the most meaningful comparison between SIMx86 and *sim-fast* can be achieved by comparing each simulation time with the native execution on the host hardware. The simulation penalty for program execution is two orders of magnitude for both SIMx86 and *sim-fast*. It is on this basis that we conclude that although SIMx86 has paid a performance penalty for its extensible design, it is still competitive with SimpleScalar *sim-fast*.

6 Related Work

Many approaches to processor simulation have been proposed in the literature. Cmelik and Kepel provide an excellent overview of several instruction set simulators and the varying technology that they use[7]. Notably absent in their categorization of tools is a distinction between trace-driven and execution-driven simulators.

The simplest method for simulating the execution of instructions is using an interpreter to decode and execute each instruction in sequence. Unfortunately, the reduced complexity of this approach often results in poor performance. Many tools have improved performance by predecoding instructions to some intermediate representation which can be decoded and executed by the interpreter at a reduced cost [3, 12, 24, 27]. SIMx86 falls in to this category of simulators.

An alternative to predecoding instructions is cross compilation. Cross compiled simulators actually use the trace file or target executable as input to a compiler which produces executable code for the host machine. Cross compilation has been successfully applied both statically [6, 10] and dynamically [5, 7, 20, 27]. However, portability is poor for cross compilation systems, since new versions of the simulator must be produced for each possible host machine.

Although many emulators and debugging tools [9, 11, 14, 17] have been developed for the Intel 80x86, relatively few instruction set simulators have been developed. SimpleScalar x86 was developed at the University of Wisconsin as a companion to the SimpleScalar simulators which were used to assess the performance of SIMx86[4]. SimpleScalar x86 is an interpreter that simulates Linux/x86 binaries on a Sparc Sun-OS host. A cache of recently decoded instructions is used in an attempt to battle performance problems that appear in interpreters. We had hoped to compare the performance of SIMx86 with that of SimpleScalar x86 since they simulate the same processor family. Unfortunately, SimpleScalar x86 is tied to a Sun-OS host machine and is also not very extensible or modifiable. Attempts to port it to other operating systems (notably Linux and Solaris) have failed to this point [1]. Thus, the decision was made to compare the performance of SIMx86 with that of SimpleScalar's *sim-fast* even though they simulate different instruction sets.

7 Concluding Remarks

We have presented the design and implementation of SIMx86, a simulator for the 80x86 family of processors. We have demonstrated the extensibility and ease of modification of our design. In addition, we have compared SIMx86 with SimpleScalar *sim-fast*[4], an extremely efficient simulator that uses a traditional approach to the design of processor simulation. Although *sim-fast* is three times faster than SIMx86, we believe that the ease of extension, modification and maintenance of our model offsets the loss of efficiency.

References

- [1] Todd Austin and Doug Burger. Personal communication, October 1996.
- [2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *Proceedings of Symposium on Principles of Programming Languages*, pages 59–70, Jan 1992.
- [3] Robert Bedichek. Some efficient architecture simulation techniques. *Winter 1990 USENIX Conference*, January 1990.
- [4] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical Report 96-1308, Wisconsin University, July 1996.
- [5] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *Proceedings of OOPSLA '89*, October 1989.
- [6] F. Chow, M. Himelstein, E. Killian, and L. Weber. Engineering a risc compiler system. *IEEE COMPCON*, March 1986.
- [7] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994.
- [8] Intel Corporation. *Pentium Pro Family Developer's Manual*. Intel Press, first edition, 1996.
- [9] DDI, Minneapolis, MN. *VIM: Virtual Machine version 1.01PD*, 1987.
- [10] Richard M. Fujimoto and William B. Campbell. Efficient instruction level simulation of computers. *Transactions of the Society for Computer Simulation*, 5(2), 1988.
- [11] P. David Gardner. WINE (windows emulator) Frequently Asked Questions. World Wide Web. <http://www.asgardpro.com>.
- [12] John Hennessy and David Patterson. *Computer Organization and Design: The Hardware-Software Interface*. Morgan Kaufman, San Francisco, 1993. Appendix A, by James R. Larus.
- [13] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, San Francisco, second edition, 1996. Appendix D: An Alternative to RISC: The Intel 80x86.
- [14] Insignia Solutions, Inc. SoftWindows Version 2.0 for UNIX. World Wide Web. <http://www.insignia.com>.
- [15] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall Software Series, 1978.
- [16] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *Proceedings of Fourth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.
- [17] James MacLean et al. DOSEMU. World Wide Web. <http://www.ednet.ns.ca/~macleajb/dosemu.html>.
- [18] B. A. Malloy. The validation of a multiprocessor simulator. *Proceedings of the 1993 Winter Simulation Conference*, pages 625–631, December 1993.
- [19] Norman S. Matloff. *IBM Microcomputer Architecture and Assembly Language: A Look Under the Hood*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.
- [20] Cathy May. Mimic: A fast s/370 simulator. *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*, June 1987.
- [21] F. H. McMahon. FORTRAN CPU performance analysis. *Lawrence Livermore Laboratories*, 1972.
- [22] M J. Philip. Performance issues for the 88110 RISC microprocessor. *Proceedings of IEEE COMPCON*, pages 163–168, February 1992.

- [23] A.J. Smith. Cache memories. *ACM Computing Surveys*, 14(3), 1982.
- [24] Rok Sosič. Dynascope: A tool for program directing. *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, June 1992.
- [25] D. A. Sykes and B. A. Malloy. The design of an efficient simulator for the pentium pro processor. *Proceedings of the 1996 Winter Simulation Conference*, December 1996.
- [26] Eric Tauck. *WASM 1.0: Wolfware Assembler for the IBM Personal Computer*. Wolfware, 1985.
- [27] J. E. Veenstra and R. J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. *Proceedings of the Second International Workshop on Modeling and Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, January 1994.
- [28] John F. Wakerly. *Microcomputer Architecture and Programming*. J. Wiley, New York, 1989.
- [29] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, first edition, 1996.

Appendix: Additional Figures

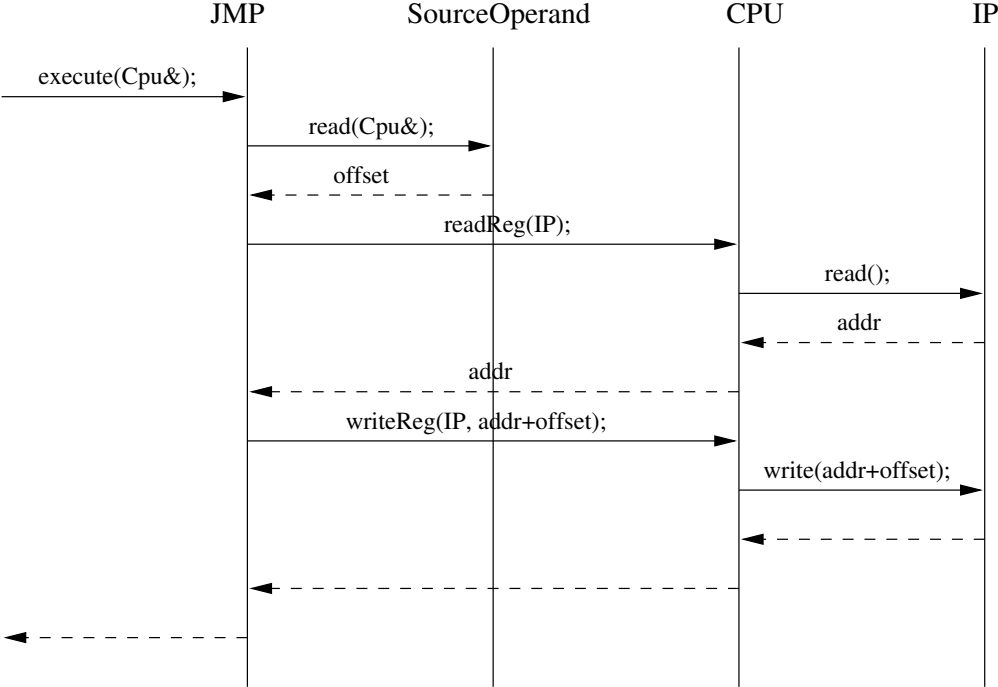


Figure 10: Object Interaction Diagram for the execution of a JMP instruction.

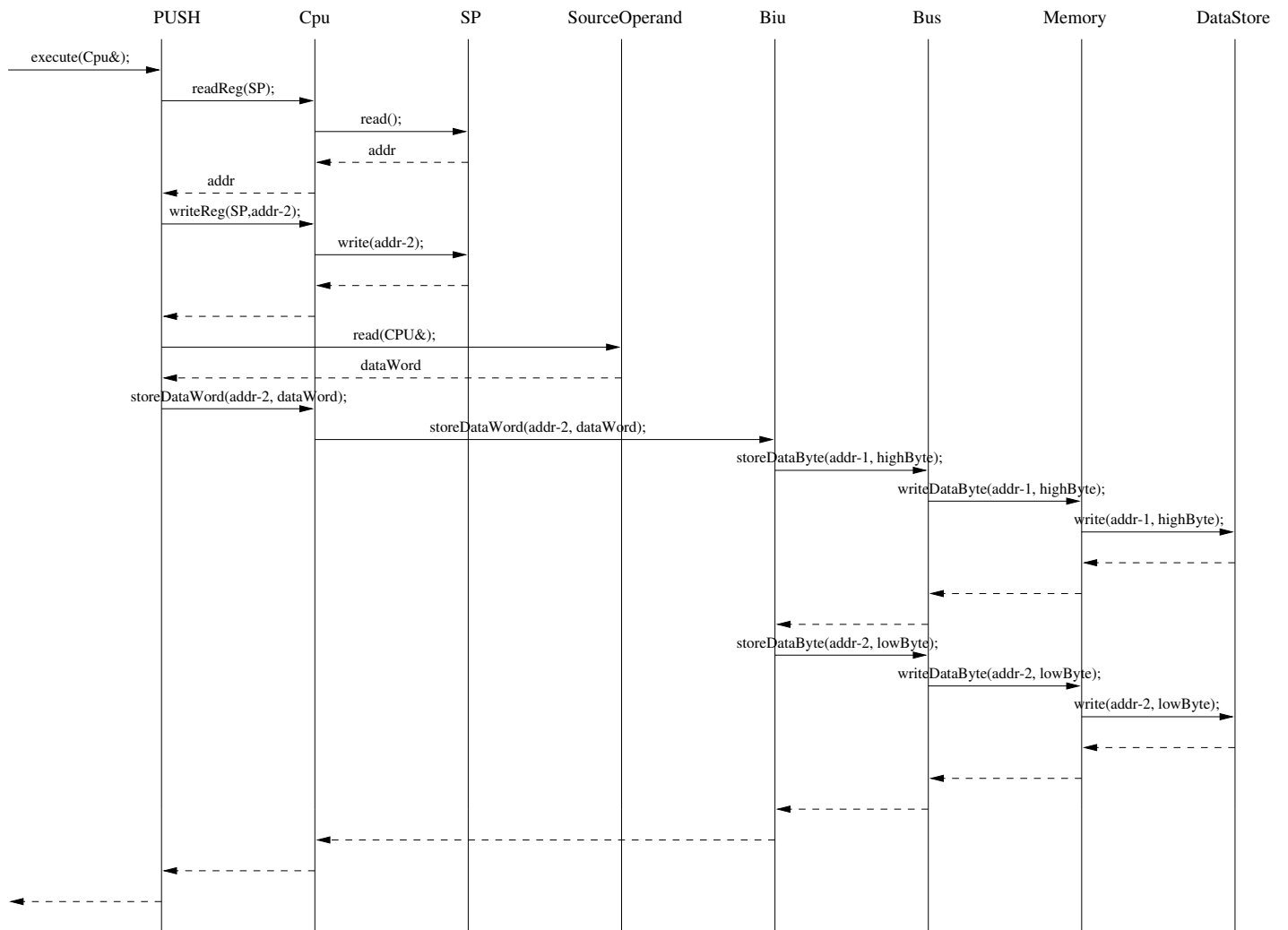


Figure 11: Object Interaction Diagram for the execution of a PUSH instruction.

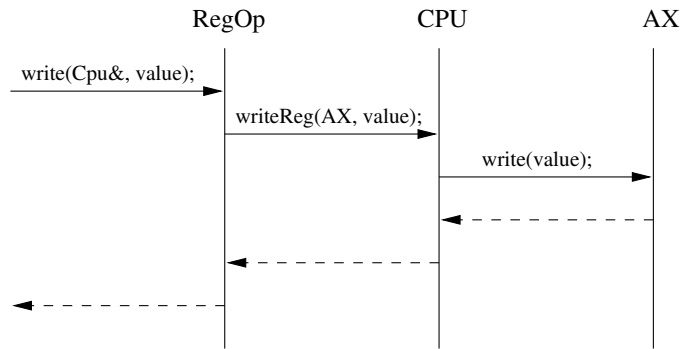


Figure 12: Object Interaction Diagram for writing to a register operand. In this example AX.

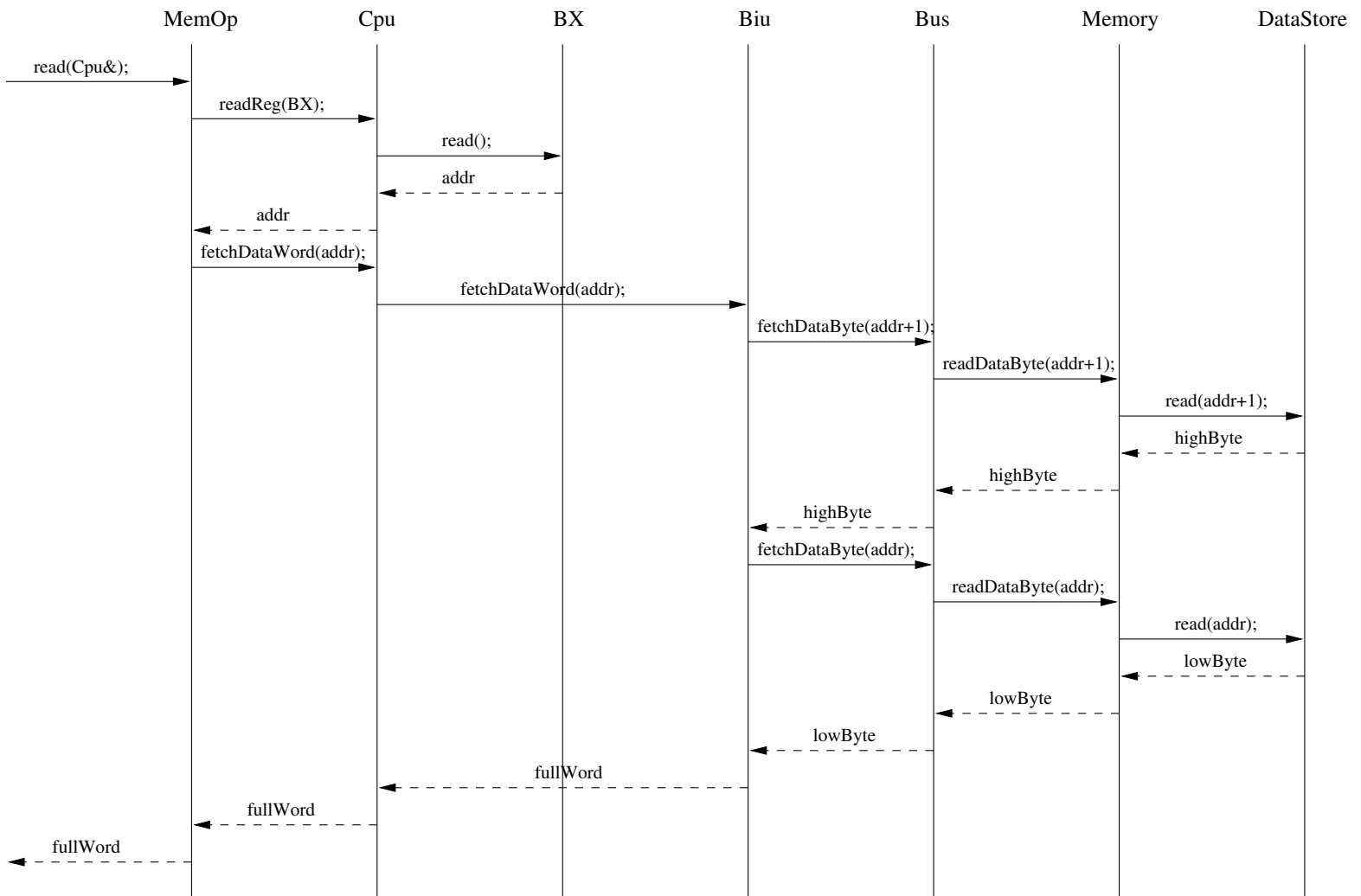


Figure 13: Object Interaction Diagram for reading a 16 bit memory operand. In this example [BX].

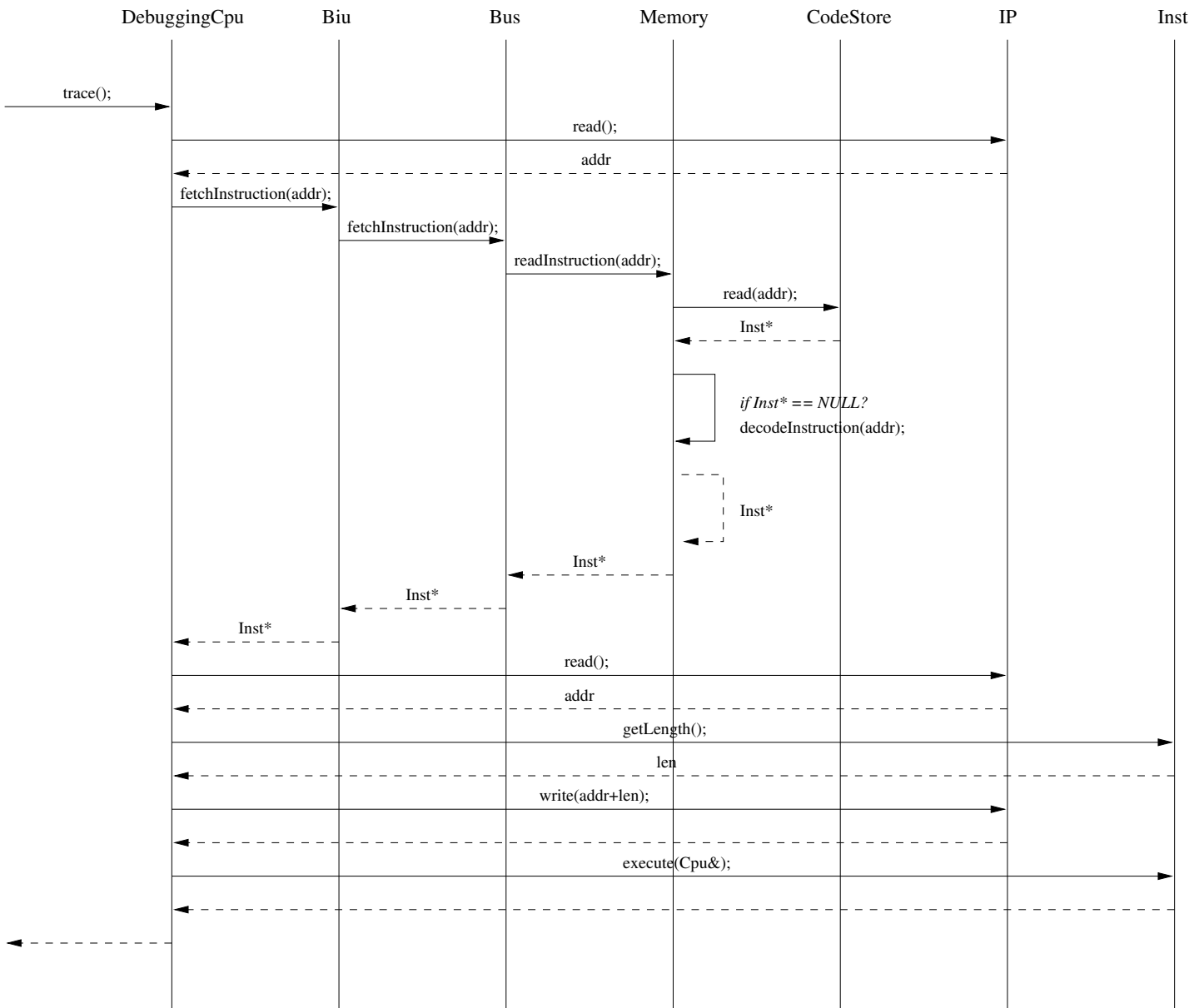


Figure 14: Object Interaction Diagram for an instruction trace.

```

> debug matmult.com
Welcome to the OO debugger.
-?
R      - Display Register Values
G      - Unconstrained Go
T      - Trace
D <Addr> - Dump Memory
Q      - Quit
-r
AX=0000 BX=0000 CX=0000 DX=0000 BP=0000 SP=FFFE SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000 IP=0100  NV NZ PL NC
0000:0100 55          PUSH BP
-t
AX=0000 BX=0000 CX=0000 DX=0000 BP=0000 SP=FFFC SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000 IP=0101  NV NZ PL NC
0000:0101 89E5          MOV  BP, SP
-t
AX=0000 BX=0000 CX=0000 DX=0000 BP=FFFC SP=FFFC SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000 IP=0103  NV NZ PL NC
0000:0103 81EC9A3A        SUB  SP, 3a9a
-t
AX=0000 BX=0000 CX=0000 DX=0000 BP=FFFC SP=C562 SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000 IP=0107  NV NZ NG NC
0000:0107 56          PUSH SI
-t
AX=0000 BX=0000 CX=0000 DX=0000 BP=FFFC SP=C560 SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000 IP=0108  NV NZ NG NC
0000:0108 57          PUSH DI
-d 0100
0000:0100 55 89 E5 81 EC 9A 3A 56 57 C7 46 FE 2D 05 31 F6
0000:0110 E9 90 00 31 FF E9 82 00 B8 35 0C F7 6E FE 99 B9
0000:0120 01 00 F7 F9 89 56 FE 8B 46 FE 99 05 00 80 83 D2
0000:0130 FF B9 00 40 F7 F9 50 B8 64 00 F7 EE 89 FA 01 D2
0000:0140 01 D0 8D 96 76 EC 01 D0 89 C3 58 89 07 B8 35 0C
0000:0150 F7 6E FE 99 B9 01 00 F7 F9 89 56 FE 8B 46 FE 99
0000:0160 05 00 80 83 D2 FF B9 00 40 F7 F9 50 B8 64 00 F7
0000:0170 EE 89 FA 01 D2 01 D0 8D 96 EE D8 01 D0 89 C3 58
-t
AX=0000 BX=0000 CX=0000 DX=0000 BP=FFFC SP=C55E SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000 IP=0109  NV NZ NG NC
0000:0109 C746FE2D05      MOV  [BP+fffe], 52d
-t
AX=0000 BX=0000 CX=0000 DX=0000 BP=FFFC SP=C55E SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000 IP=010E  NV NZ NG NC
0000:010E 31F6          XOR  SI, SI
-g
Program terminated normally
>

```

Figure 15: Using the debugger interface.