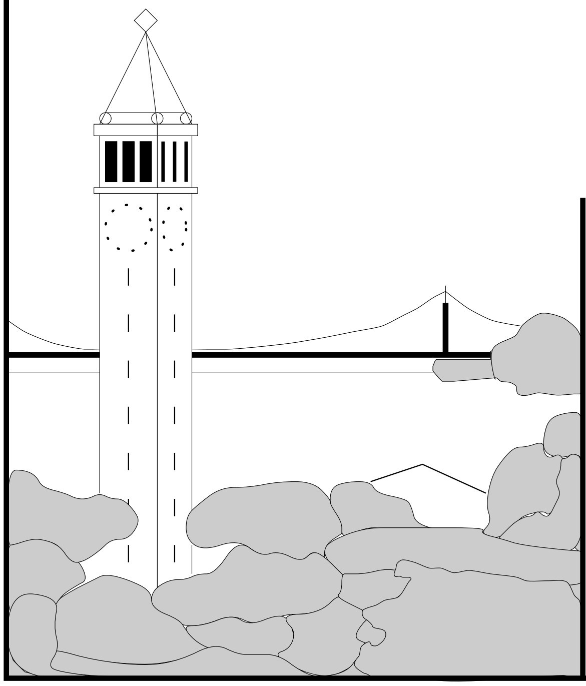
Design and Implementation of the IRAM Architecture Manual and Functional Simulator

David R. Martin



Report No. UCB/CSD-98-1025

December 1998

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

[1] In a microprocessor project such as the Berkeley Intelligent RAM (IRAM) Project, there needs to be a golden architectural model that is simple, precise, and verifiable. For these reasons, the golden model is written as a computer program so that it can be compared to other models (e.g. RTL or gate-level) in an operational manner. Furthermore, the architectural model is often used for compiler, operating system, and application development, and consequently needs to be very fast. Thus, fast languages such as C++ or even assembly are common choices. In addition to a fast simulation environment, developers need good documentation. In a microprocessor project, the documentation needs to be up-to-date and correct with a high degree of confidence. This paper describes the approach taken in the IRAM project to derive the architectural simulator and architecture manual from a single source. This method disallows many types of inconsistencies between the model and the documentation of the model that can remain undetected in traditional approaches.

1 Introduction

The IRAM vector architecture [7] is a MIPS-IV coprocessor architecture that extends MIPS with vector processing. Although the number of instructions is not larger than one would find in the computational core of a traditional RISC architecture, the number of opcodes is quite large due to the orthogonality of the instructions across several independent axes: data width, data type, speculation, predication, and source register specification. The size and highly regular nature of the architecture leaves open the opportunity for typographical errors in the specification and implementation of the simulation tools.

In order to reduce the chance for inconsistency between the models and the documentation of the models in the IRAM project, we bring an old solution to a new context. Literate programming [8, 3] was a method pioneered by Knuth as a means of writing highly readable, or "literate" programs. The goal was to create well documented software and a wholly better approach to software development. Superior software development methods yield more reliable and more maintainable software. As a demonstration of the first literate programming system `web` [2], the document processing system TeX [4, 6, 5] was originally written as a literate program.

Literate programming did not prevail as common practice, though it enjoys the following of die-hard fans and remains a good teaching tool. This paper does not present an argument to revive literate programming practices in general. Rather, it presents one application area where literate programming is a superb method: microprocessor architecture manuals. The emphasis of literate programming – by virtue of its name alone – is programming. That is, a literate program is a program that happens to be easy to read and understand. We turn around the emphasis in order to produce a document that also happens to be a program. Thus, the IRAM architecture manual can be compiled into a functional architectural simulator.

Most microprocessor architecture manuals contain a great deal of pseudo-code. For example, the MIPS RISC Architecture manual [1] contains, in appendices, detailed descriptions of all MIPS instructions. Usually one, but often several pages are dedicated to each instruction. On these instruction definition pages, every instruction contains an "Operation" subsection that details the operation of the instruction using formal symbolic notation rather than plain English. The goal is to use a precise language in which to describe the instructions, since English is not precise enough. We simply bring this argument to its logical conclusion. Why not use executable code to describe each instruction? As a result, the architecture manual would describe the instruction set in as precise a manner possible – in a true operational manner.

Before we describe the details of the IRAM architecture manual, Section 2 describes literate programming with a small example. Section 3 describes the design and implementation of the IRAM manual/simulator. Section 4 presents the performance of the IRAM simulator compared to a traditional approach. Section 5 discusses the costs of our approach and the advantages and disadvantages of using the C++ language in our implementation. Section 6 concludes the paper.

[1] This work was supported by DARPA grant #DABT63-96-C-0056.

2 Literate Programming

Literate programming is a technique of mixing code and documentation pioneered by Don Knuth with the `web` system. Since its inception, `web` has inspired the creation of a variety of literate programming tools. The tool we used is called `noweb` [8]. `noweb` targets the LaTeX document formatting system, and is independent of programming language. The basic idea of literate programming is to merge a program and its documentation. Figure 1 shows how the two objects of interest – the program and the technical manual – are derived from a single parent document: the `noweb` document. This mechanism is best described with a small example.

A `noweb` document consists of alternating documentation and code chunks. Figure 2 shows the example `noweb` document. The example contains 5 code chunks. Code chunks begin with "⟨⟨code chunk name⟩⟩=" and end with "@", and may contain arbitrary C++ code. Each code chunk is separated by a possibly empty documentation chunk. The documentation chunks may contain arbitrary LaTeX commands.

A code chunk can be viewed as a simple macro. It may be extracted from the file by `noweb` and it may be included by any number of other code chunks in the `noweb` file. In Figure 2, the `globals` code chunk is included by the `example.cc` code chunk. Figure 3 shows the result of asking `noweb` to extract the `example.cc` chunk from the `noweb` source file. Note that `noweb` preserves indentation for included code chunks for good readability.[2]

Figure 4 shows the result of passing the `noweb` document through `noweb` and LaTeX to yield the literate program. Note that not only did `noweb` format the code chunks nicely via LaTeX formatting commands, but it also added many useful cross-references. Both code chunks and identifiers are cross-referenced to allow easy navigation of the code. Code chunks are given numbers such as "⟨*includes 1d*⟩" to denote that the "*includes*" code chunk is the fourth (d = 4) code chunk on page 1. Identifiers are also cross-referenced. For example, we see that the "`exit`" identifier is defined by chunk 1d, and is used in chunk 1b. Since the `noweb` tool knows nothing about the programming language (C++ in this case), it relies on the "`%def`" commands to determine which identifiers a chunk defines. `noweb` then automatically finds uses of identifiers in all other code chunks in the document.

Note that the `includes` code chunk seems to be defined twice. The chunk is not being redefined, however. `noweb` concatenates all instances of a chunk, allowing a code chunk to be conveniently defined in multiple pieces. It was convenient to separate the two `#include` statements in their own code chunks so that it is clear that `stdlib.h` defines `exit`, and `stdio.h` defines `puts`.

3 The IRAM Simulator and Manual Implementation

The goal in this work was to produce a traditional-looking architecture manual whose "pseudo-code" could be compiled into an architectural simulator. Writing executable code that looks like pseudo-code and can be compiled into an efficient program is extremely challenging. Pseudo-code can be made concise and clean precisely because it does not need to be compiled and it does not need to be computationally efficient. In sum, the manual/simulator needed to satisfy the following opposing requirements:

- Instruction operation definitions were to be written using a high-level pseudo-code-like symbolic notation.
- Instruction operation definitions needed to be written in an existing language for which good compilers exist, and in a style that allows complier optimizations.

Several languages were initially candidates: Java, Modula-3, and C++. These languages were chosen because object orientation, exceptions, and operator overloading were deemed important to create code that looks like pseudo-code. It was clear that an object-oriented language was required for data abstraction purposes. All three languages have good support for exceptions, though the C++ model is the weakest. Since operator overloading provides a

[2] `noweb` will pass file and line information via C preprocessor directives so that the compiler and debugger reference the original `noweb` source file instead of the intermediate C++ source file.

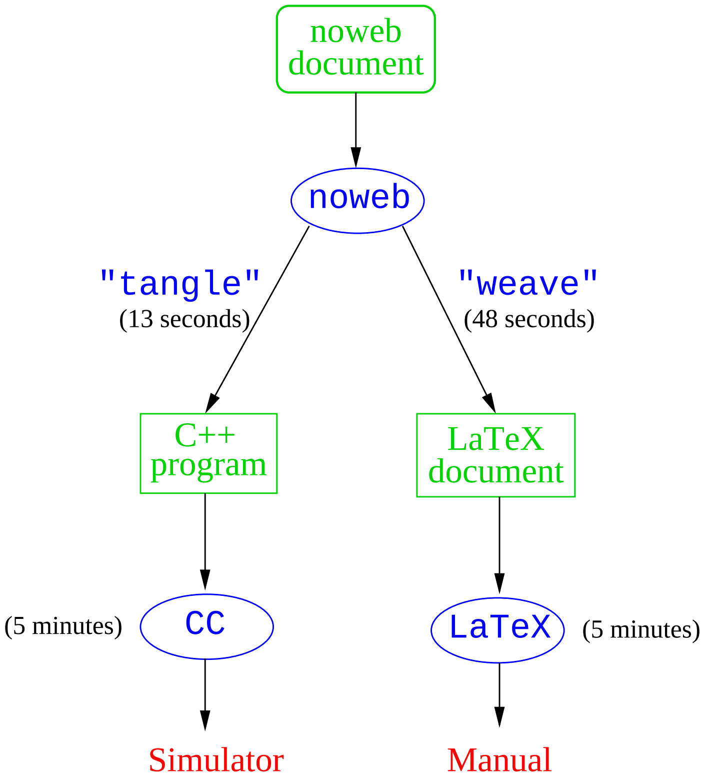


Figure 1: Literate Programming Figures 2, 3, and 4 show an example `noweb` document at three points in the tool flow. Figure 2 shows the `noweb` document. `noweb` extracts the $\texttt{C}^{++}$ program shown in Figure 3. `noweb` also produces a LaTeX document, the rendered version of which is shown in Figure 4. The process of producing a computer program from a literate program is called *tangling*. The process of producing the readable document is called *weaving*. Execution times are shown for the full IRAM simulator/manual. The MIPSpro 7.2.1 $\texttt{C}^{++}$ compiler was run on a 180 MHz MIPS R10000 system; `noweb` and LaTeX were run on a 300 MHz Pentium II system.

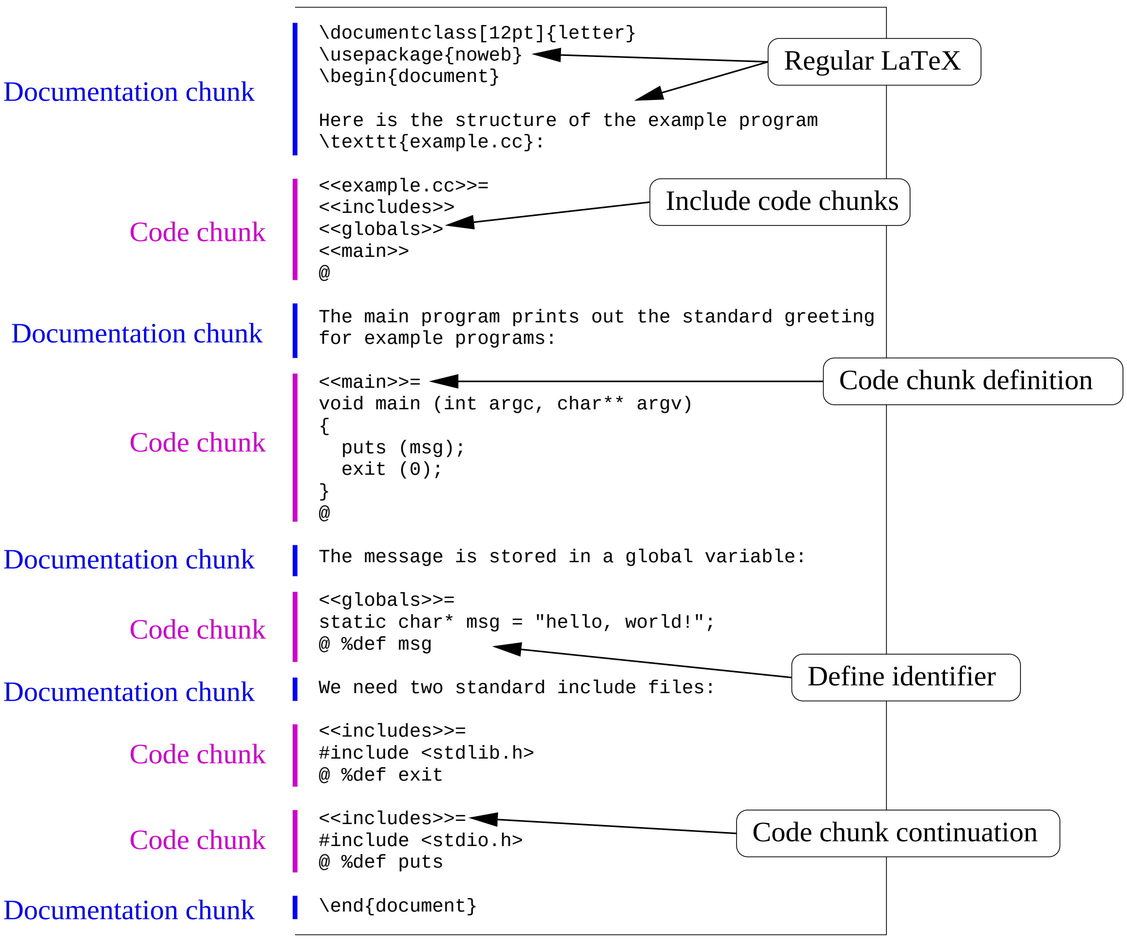


Figure 2: A `noweb` example. This is the `noweb` document that is written by the user. It consists of alternating code chunks and documentation chunks. The code chunks collectively form a valid `C++` program, while the documentation chunks form a valid LaTeX document.

```
#include <stdlib.h>
#include <stdio.h>
static char* s_pc_msg = "hello, world!";
void main (int i_argc, char** ppc_argv)
{
  puts (s_pc_msg);
  exit (0);
}
```

Figure 3: `C++` program extracted from the `noweb` example document. This is the result of passing the `noweb` source of Figure 2 through `noweb` to extract the embedded `C++` program.

Here is the structure of the example program `example.cc`:

1a ⟨*example.cc* 1a⟩≡
⟨*includes* 1d⟩
⟨*globals* 1c⟩
⟨*main* 1b⟩
This code is written to file `example.cc`.

The main program prints out the standard greeting for example programs:

1b ⟨*main* 1b⟩≡

```
void main (int i_argc, char** ppc_argv)
{
  puts (s_pc_msg);
  exit (0);
}
```

Uses `exit` 1d, `puts` 1e, and `s_pc_msg` 1c.

The message is stored in a global variable:

1c ⟨*globals* 1c⟩≡

```
static char* s_pc_msg = "hello, world!";
```

Defines:
`s_pc_msg`, used in chunk 1b.

We need two standard include files:

1d ⟨*includes* 1d⟩≡

```
#include <stdlib.h>
```

Defines:
`exit`, used in chunk 1b.

1e ⟨*includes* 1d⟩+≡

```
#include <stdio.h>
```

Defines:
`puts`, used in chunk 1b.

1

Figure 4: Rendered document extracted from the example `noweb` document. The `noweb` source of Figure 2 is first passed through `noweb` to produce a LaTeX document (not shown). This figure shows the rendering of that LaTeX document.

means of writing high-level syntax, and since neither Java nor Modula-3 provide operator overloading, C^{++} seemed the best choice. In addition, high-quality C^{++} compilers are abundant in comparison to Java and Modula-3 compilers. The need to interface with other tools and libraries provided yet more reasons to use C^{++}.

The abundance of features in the C^{++} language was attractive for this project, but great care was taken in their use. It is not difficult to employ a wide array of C^{++} constructs to produce an undeciferable program. Furthermore, many layers of dynamically-dispatched abstractions could yield high run-time overhead. Although I explored a wide array of available mechanisms, the only "exotic" features that I use in the final simulator is operator overloading. In addition, I employed exclusively static data abstractions. There is no inheritance, and consequently no dynamic dispatch. In theory, since all method invocations can be determined at compile time, the compiler could completely remove them through inlining and standard code optimizations. Thus, through carefully chosen constructs, highly readable and efficient code can be written in C^{++}.

Before we present the implementation of the IRAM manual, let us consider an example from page A-37 of the MIPS Architecture manual [1]. The top portion of Figure 5 shows the "Operation" section for the Branch On Not Equal (BNE) instruction. For each instruction, the "Operation" section presents pseudocode that defines the instruction in a precise manner. If we apply the techniques we used to write the IRAM manual to this example, we would have written the code shown at the bottom of Figure 5. We may then take this chunk of C^{++} code and drop it into a context that defines the free variables such as `GPR` and `rs`. We will then have used the instruction definition in the architecture manual to build an architectural simulator.

The conversion of the pseudo-code in Figure 5 into executable C^{++} code was straight-forward, and yielded very efficient code. Furthermore, the code remains high-level and extremely easy to read. We have lost the ability to use fancy mathematical symbols, but we have gained the ability to execute the code chunk and verify its correctness against other models. Extracting the simulator from the architecture manual improves both the accuracy of the pseudo-code and the consistency between the pseudo-code and the simulator.

3.1 The Role of `noweb`

The goal of this project was to write the MIPS vector instruction set extensions in the manner of Figure 5. All instruction code chunks could then be collected and compiled into a fast architectural simulator. The translation from pseudo-code to C^{++} is simple for the MIPS instruction set, but is more difficult for the vector instructions of the IRAM project. Vector instructions represent a large amount of computation, and are more complex than most scalar instructions. For this reason, it was a challenge to write concise and efficient C^{++} code for our vector instructions, but the goal was attained.

Figure 6 shows an example page from the IRAM architecture manual. The operation of the unsigned vector add instruction is defined by the following code chunk:

```
<<check vector length>>
<<for each unmasked vp>> {
  if (vv) {x = VR[src1][vp]; y = VR[src2][vp];}
  if (sv) {x = VSR[src1]; y = VR[src2][vp];}
  VR[dest][vp] = x + y;
}
```

This chunk of code handles all 4 versions of the unsigned vector add instruction: `vadd.u.sv`, `vadd.u.vv`, `vadd.u.sv.1`, and `vadd.u.vv.1`. The "`.sv`" qualifier means that the first source is scalar and the second vector. The "`.vv`" qualifier means that both sources are vectors. The "`.1`" qualifier means that the instruction uses the alternate mask. It should be clear from context that the `alternateMask`, `vv`, and `sv` variables are true when the relevant qualifier is in effect. This code chunk uses two other code chunks, which are not shown in the figures. Expanding these two chunks, we get:

```
if (VCR[vc_vl] > VCR[vc_mvl])
  throw Exception (EvectorLength);
for (vp = 0; vp < VCR[vc_vl]; vp++)
  if (alternateMask
      ? VFR[vf_mask1][vp]
      : VFR[vf_mask0][vp]) {
    if (vv) {x = VR[src1][vp]; y = VR[src2][vp];}
    if (sv) {x = VSR[src1]; y = VR[src2][vp];}
    VR[dest][vp] = x + y;
}
```

$$
\begin{array}{ll}
\mathrm{T}: & \mathrm{target} \leftarrow (\mathrm{offset}_{15})^{14} \parallel \mathrm{offset} \parallel 0^2 \\
 & \mathrm{condition} \leftarrow (\mathrm{GPR[rs]} \neq \mathrm{GPR[rt]}) \\
\mathrm{T}+1: & \text{if condition then} \\
 & \quad \mathrm{PC} \leftarrow \mathrm{PC} + \mathrm{target} \\
 & \text{endif}
\end{array}
$$

⟨*execute BNE: branch on not equal*⟩≡

```
/* T:   */  target = signExtend (offset << 2);
            condition = (GPR[rs] != GPR[rt]);
/* T+1: */  if (condition) {
              PC = PC + target;
            }
```

Figure 5: Nowebifying the MIPS Architecture Manual The top portion shows the definition of the BEQ instruction, as shown on page A-37 in the MIPS Architecture Manual [1]. The bottom portion shows how this code would be written as a literate program. We have lost the fancy mathematical symbols (like $\leftarrow$, $\|$, and $\neq$), but we have gained the ability to use this piece of code in an architectural simulator.

This type of macro expansion allows code reuse. A principle of software engineering is that code should never be replicated, since replicating code will replicate bugs. The `noweb`-style macros allow single pieces of code to be replicated safely, since the code is reused without being manually replicated by the programmer. The number of lines of `C`$^{++}$ code in the simulator `noweb` document is 7229. Tangling this document produces a 17757 line `C`$^{++}$ program. Thus, `noweb` reduced the number of lines of code that I had to write by a factor of 2.3. If one believes that bugs are a linear function of the number of lines of code, as is commonly accepted, then using `noweb` should have reduced the number of bugs in the simulator. This effect was seen in practice. Since the instruction definitions share a tremendous amount of code, debugging one instruction had the effect of also debugging all other similar instructions.

Note that in writing instruction definitions, we have made an assumption that the reader will understand the type and meaning of certain variables according to context. These variables must, of course, be declared and defined elsewhere, but the instruction definition page is not the appropriate place. The fact that we can rely on the reader to provide some amount of meaningful context and common sense is vitally important to keeping pedantic details (such as variable declarations) out of the executable pseudocode. Both `noweb` and `C`$^{++}$ are exceptionally useful in providing the context to the compiler that the reader provides naturally. `noweb` provides the mechanism that drops the instruction operation code chunk into a context that defines all of the free variables for the compiler. `C`$^{++}$ provides the context that defines the undefined operators.

3.2 The Role of C^{++}

Operator overloading is easy to misuse, since one can cause arbitrary side-effects through the use of seemingly innocuous operators like the plus sign. In order to avoid this situation, I strictly adhere to the rule that every overloaded operator does exactly what you think it is doing. This rule may seem imprecise, but is sufficient for producing readable code in practice. For example, the line that reads "`VR[dest][vp] = x + y`" in Figure 6 contains 4 implicit calls to overloaded operator functions: both brackets operators, the assignment operator, and the plus operator.

85

Unsigned Vector Integer Add — VADD.U

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\texttt{VADD.U}\begin{cases}\texttt{.VV[.1]}\ \texttt{vr}_{\texttt{dest}},\ \texttt{vr}_{\texttt{src1}},\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.SV[.1]}\ \texttt{vr}_{\texttt{dest}},\ \texttt{vs}_{\texttt{src1}},\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

85 ⟨*execute unsigned vector integer add* 85⟩≡

```
⟨check vector length 76b⟩
⟨for each unmasked vp 75c⟩ {
  if (vv) { x = VR[src1][vp]; y = VR[src2][vp]; }
  if (sv) { x = VSR[src1]; y = VR[src2][vp]; }
  VR[dest][vp] = x + y;
}
```

Uses `VR` 226b and `VSR` 226b.
This code is used in chunk 251.

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the unsigned integer sum of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`EvectorLength`	Vector length too long.

Notes

The only difference between `VADD` and `VADD.U` is that `VADD` can overflow, while `VADD.U` cannot.

Figure 6: Example page from the IRAM Architecture Manual. This instruction performs unsigned vector integer addition. The "Operation" section shows that a scalar addition is performed for each virtual processor (VP), or vector element. Both sources may be vector registers (specified by the `.vv` qualifier), or the first source may be a scalar value (specified by the `.sv` qualifier). The instruction is always masked, and the `.1` qualifier specifies that the alternate mask is in effect. See the text for expansions of the "⟨check vector length 76b⟩" and "⟨for each unmasked vp 75c⟩" code chunks.

Code Description	Lines	Percent
Code on instruction definition pages	806	11
Overloaded operator functions	806	11
Utility functions	1026	14
Register file abstractions	355	5
Instruction dispatch	4238	59
Total	7231	100

Figure 7: Breakdown of Simulator Code The `C++` code from the simulator `noweb` document is divided into 5 categories. The reason that the dispatch code is so large is that though there are only 116 vector instructions, there are 1082 vector opcodes. The size of the dispatch code is a linear function of the number of opcodes.

There is nothing deceptive going on behind the operators, however. The plus operator performs integer addition, the brackets dereference an (abstract) 2-dimensional array, and the assignment operator performs assignment.

Why, then, are these operators overloaded? For various technical reasons not important to this discussion, the vector register file `VR` cannot be implemented as a simple array, since it can hold values of many different data types. Furthermore, the innocuous plus operator is actually performing either 8-bit, 16-bit, 32-bit, or 64-bit integer arithmetic according to the contents of a vector control register. Thus, operator overloading allows us to multiplex the same simple code chunk over a large group of subtly different but basically identical instructions.

In sum, I use `C++` to provide register file abstractions that look like simple arrays, and overloaded operators to hide the details of arithmetic. The majority of the complexity of an architecture lies in the control flow rather than the computation. For example, although it is important to precisely define two's complement addition somewhere in the simulator, the instruction definition page is not a good place. It is far more important how data and control flow are affected by the instruction. Most of the instruction definition code in the IRAM manual deals with detecting and processing exceptional conditions. That the complexity of an architecture lies in its control flow is the overriding rule that guided decisions about what should be exposed and what should be abstracted in the instruction definition code chunks.

3.3 Code Breakdown

Figure 7 shows the breakdown of `C++` code in the simulator `noweb` document. Much of the code is rather trivial, as it deals with dispatching the 1000 odd opcodes to instruction execution routines. After excluding the dispatch code, only 40% of the code is non-trivial. A total of 806 lines (11%) are directly visible to the reader in instruction definitions such as Figure 6. This yields an average of 7 lines of code per instruction definition code chunk. The range is 1 line to 19 lines, but Figure 6 is representative.

Only 255 lines (5%) of code were required to implement the 5 register file abstractions.[3] This code is extremely simple, easy to debug, and easy to maintain. The overloaded arithmetic operators consume 806 lines (11%) of code. This code is very regular, and also easy to implement. Due to the restrictions of `C++`, I was not able to use operators for all arithmetic operations. There are 1026 lines (14%) of code devoted to utility functions such as the `signExtend` function in Figure 5.

Overall, the code is not complex, and is quite short. Since this simulator is the reference model for our research project, it is important that it is free of bugs and easy to change. Due to their simplicity, the layers of abstraction introduced by the register file abstractions and overloaded operators do not compromise this requirement.

[3] In addition to the scalar integer register file, there are 4 vector register files.

Benchmark	Version	vops	inst	ops	Description
vecadd	scalar	0	8.7M	8.7M	Strip-mined memory-to-memory vector add of length `N` vectors.
	vector	6M	1.4M	7.4M	
mvmult	scalar	0	5.9M	5.9M	Product of an `N x N` matrix and a length `N` vector.
	vector	3M	0.6M	3.6M	
cromakey	scalar	0	9.4M	9.4M	Select pixels from one of two images based on the values in one of the images.
	vector	5M	1.1M	6.1M	

Figure 8: Benchmarks The scalar versions were written in plain C. The vector versions were written in vector assembly language. `vops` is the number of scalar operations performed by the vector instructions. `inst` is the total number of scalar and vector instructions executed. `ops` is the total number of scalar operations, including those performed by vector instructions. For example, a scalar instruction contributes 0 vops, 1 inst, and 1 op; a vector instruction of length N contributes N vops, 1 inst, and N ops.

4 Performance Analysis

We performed three experiments in order to understand the performance of simulating vector instructions with our methods. Figure 8 describes the three benchmarks we used in the 3 experiments. The following sections describe the experiments.

4.1 Scalar vs. Vector Performance

Two versions of each benchmark in Figure 8 were coded – one in plain C ("scalar"), and one in vector assembly language ("vector"). The C versions were compiled with aggressive compiler optimizations. Since the kernels are all small and simple, the compiler's optimized non-vector assembly language is a fair comparison to hand-coded vector assembly language.

The simulation methods described in this paper were applied only to simulating vector instructions. For all scalar instructions, we use the fast MIPS-V simulator MINT+, written by Jack Veenstra. Thus, though we send both scalar and vector versions of the benchmarks though our simulator, pure scalar code is simulated entirely by MINT+.

Figure 9 shows the difference in simulator execution time for scalar and vector versions of each benchmark. The total amount of computation is held constant between versions of the same benchmark. We see that there is no penalty for simulating vectorized codes. The vectorized code simulates 19% slower to 27% faster than functionally equivalent scalar code. This data demonstrates that our simulator simulates vector instructions at a speed competitive with the very fast simulation of pure scalar code by the MINT+ simulator.

4.2 Vector Performance

Though Figure 9 shows that the speed of simulating vector instructions is good, it does not show that the simulation could not be faster. In order to determine if our simulation methods carry performance penalties, we need to compare our vector simulator to others. Given that vector architectures are not widespread, it is difficult to make a meaningful comparison between our and other simulators. Even if there were another vector architecture simulator available, a head-to-head comparison would tell us little about the overhead of writing highly readable code, isolated from all the other differences of the two simulators.

To see how architectural factors can affect simulation speed, consider our vector floating-point compare instruction. This instruction consists of 144 opcodes, accounting for speculation, data-type, predicate, source-register specification, and mask specification. The most highly optimized simulator would perform a table lookup on the opcode to branch to one of 144 virtually identical routines that implement this one instruction. Such a style of writing a simu-

Benchmark	Version	normalized execution time
vecadd	scalar	0.84
	vector	1.00
mvmult	scalar	1.37
	vector	1.00
cromakey	scalar	1.03
	vector	1.00

Figure 9: Scalar vs. Vector Simulator Performance Times are normalized within each benchmark to the time for the vector version. For the same computational task, vectorized code simulates 19% slower to 27% faster than scalar code. The simulator has a slowdown of 100 for scalar code on a 180MHz MIPS R10000 system with a unified 2MB cache, using -O2 optimization with the MIPSpro 7.2.1 $\mathtt{C}^{++}$ compiler.

Benchmark	normalized execution time		
	vsim-0	vsim-1	vsim-2
vecadd	1.30	1.09	1.00
mvmult	1.30	1.11	1.00
cromakey	1.46	1.11	1.00

Figure 10: Vector Simulator Performance Times are normalized within each benchmark to the vsim-2 simulator. *vsim-1* is uniformly 10% slower than *vsim-2*. *vsim-0* is 20% to 30% slower than *vsim-1*. Figure 11 describes the differences between the *vsim-0*, *vsim-1*, and *vsim-2* simulators.

Simulator	Description
vsim-0	The simulator is compiled as a literate program, as shown in Figure 6.
vsim-1	All operator overloading is removed from the *vsim-0* simulator.
vsim-2	Hand-optimizations are added to the *vsim-1* simulator. These optimizations primarily involve loop fusion and moving switch statements out of inner loops.

Figure 11: Description of Simulators of Figure 10 Each benchmark is executed on three different implementations of the architectural simulator. The higher vsim numbers consist of less readable, less maintainable, and more hand-optimized code.

lator is time-consuming, error-prone, and difficult to maintain. It is more likely that the author would attempt to strike a balance between specializing the routines that implement instructions, and merging similar routines to reduce programming effort.

Writing a single routine that implements all 144 opcodes is not difficult if one uses data abstraction with objects and operator overloading as we did. The performance cost results from the fact that branches are introduced into the inner-loops of vector instructions that could have been removed in a more brute-force implementation. Note that this is a problem only because we have instructions that are orthogonal across a large number of axes, and because this is a vector instruction set where each instruction contains a loop.

In order to isolate the methods of our simulator implementation from artifacts of the IRAM vector architecture, we compare several implementations of our vector simulator to each other. This allows us to completely eliminate vector architectural artifacts from the analysis. Figure 11 describes the three simulators we compared with the vectorized kernels. The first simulator, *vsim-0* is the literate simulator with all data abstraction and operator overloading as shown in the example page in Figure 6. The next simulator, *vsim-1* removes all operator overloading from the simulator. As discussed in Section 3, there is a great deal of operator overloading in *vsim-0*. Finally, *vsim-2* adds many hand-coded optimizations to *vsim-1*. These optimizations are of the nature discussed earlier in this section that allow each minute variant of an instruction to be highly optimized, so that the inner loop contains no branches. Note that I was careful to write *vsim-0* so that the compiler would have the opportunity to perform the optimizations that are represented by *vsim-1* and *vsim-2*.

Comparing *vsim-0* to *vsim-1*, we see that operator overloading and data abstraction cost 20-30% in simulator execution time. I believe this to be an acceptable cost, considering the benefits of the approach. This number was initially 200-300%, until I removed a seemingly innocent pointer indirection from a critical code path. Microbenchmarks showed an 1800% performance penalty for a naive 1-dimensional array abstraction; removing the guilty pointer indirection reduced this penalty to a mere 15%.[4] It is disappointing, but perhaps not surprising, that the compiler cannot reduce the penalty of such abstractions to 0%. Since array abstractions are used for the register files and are consequently abundant in our code, an overall penalty of 20-30% for using both 1-D and 2-D overloaded array syntax is not unreasonable.

Comparing *vsim-1* to *vsim-2*, we see that simulating similar opcodes with a single routine costs 10% in simulator execution time. Though, in theory, the compiler could perform the optimizations to reduce this overhead to zero, it is not surprising that the optimizer did not find the optimal solution. The *vsim-2* simulator represents optimal C++ code for very long code sequences. A 10% performance penalty for implementing tens or even hundreds of instructions with single code sequences is far within the bounds of being acceptable.

The approach of using literate programming to write the architecture manual/simulator is shown in the *vsim-0* to *vsim-1* comparison, and is independent of the slowdown shown between *vsim-1* and *vsim-2*. Although the *vsim-2* implementation would be desired for performance reasons, it would only be practical if the architecture were static. Because the IRAM architecture is not frozen, we need the ability to change instructions easily. A performance penalty of 10% to reduce the amount of code that implements the instructions by an order of magnitude is certainly a good trade-off. Thus, the cost of the literate programming approach, using C++ overloaded operators and object abstractions costs 20-30% in simulator execution time, as shown by the *vsim-0* to *vsim-1* comparison.

4.3 Compiler Optimization

Since it was an assumption in this work that we could rely on the compiler to perform many optimizations, it was interesting to quantify the effect of compiler optimization. Figure 12 shows that for scalar code, simulator execution time shows 2.3-2.6x speedups, while simulating vector code shows 10-13x speedups. Since we wrote the routines that simulate vector instructions using many abstractions that we expected the compiler to remove, an order of magnitude speedup from compiler optimization was expected.

[4] A proper 2-dimensional array abstraction costs 480% per array access (e.g. `VR[src][vp]`).

Benchmark	Version	normalized execution time -O0	-O2
vecadd	scalar	2.60	1.00
	vector	13.00	1.00
mvmult	scalar	2.30	1.00
	vector	11.40	1.00
cromakey	scalar	2.60	1.00
	vector	10.50	1.00

Figure 12: Effect of Compiler Optimization on Simulation Speed Times are normalized within each row to the *-O2* execution time. For scalar code, compiler optimization yields 2.3x to 2.6x speedups. For vector code, however, compiler optimization yields 10x to 13x speedups. These large speedups for the vector code are expected, since the code that simulates vector instructions was written with many overheads that we expected the compiler to remove through optimization.

I have noted that it was important to avoid pointer indirections in order to reap the benefits of compiler optimization. At one point in this work, the speed of *vsim-0* was much slower than is reported here. Eliminating an seemingly innocent pointer indirection immediately yielded a 2-3x improvement in simulator execution time. Though the MIPSpro compiler we used is excellent, the 10-13x speedup from optimization was not the result of applying a heroic compiler to arbitrary code. In contrast, the simulator was implemented very carefully in order to enable extensive compiler optimization.

5 Discussion

The goal of this work was to produce a literate IRAM architecture manual that could be compiled into a fast architectural simulator. This goal was met, with a simulator performance penalty of 20-30%. However, it is both interesting and important to ask if the end justified the means. If this style of programming is extremely difficult, then the advantages of owning the end-product are diminished.

The concepts of literate programming are extremely simple. The length of Section 2 is representative of many tutorials on the subject. Thus, issues pertaining to the use of literate programming in this project consumed little of my time. The vast majority of the effort was spent in deciding how best to use C^{++} to reach the goal of efficient, high-level syntax. The simple and elegant solution was preceded by several relatively ugly implementations that did not offer additional functionality. This startup cost of learning and discovery cannot be fairly included in an evaluation of the cost of this method. We must therefore estimate the cost of setting up the proper data abstractions after only one or two iterations, rather than five or six.

Given that the final solution was far simpler than initial efforts, I postulate that the cost of coding the proper operators and objects is quite low. After the infrastructure of high-level syntax was complete, extremely complex instructions could be coded quite clearly in 10s of lines of code. Even accounting for the overhead of implementing an infrastructure that provides high-level syntax, the total number of lines of code in the simulator is reduced. A smaller simulator is easier to debug and easier to maintain. High-level programming is also less prone to error, since low-level mechanisms are hidden – and protected – behind objects and operators. Furthermore, since nearly all instructions shared the low-level code that implements computational elements and register files, only a small number of instructions needed to be debugged before nearly all instructions became operational. The benefits of high-level programming are well known, and they apply in full force to this application.

This project relied heavily on some less frequently used features of the C++ language such as operator overloading. As a result, I spent much time both evaluating the features available for the task at hand and wishing that certain other features were available. The following sub-sections discuss the language features that were helpful in this work, the language features that were avoided, and language features that would have been helpful if they existed in C++.

5.1 Helpful Language Features

I chose C++ as the language for this project for many reasons. The following is a list of the most important features:

Objects Data abstraction is important in nearly every program including the IRAM simulator. Objects provided an extremely natural way to abstract the register files, memory, and low-level data types that the reader does not want to know about.

Operator Overloading This completes the illusion that the expression `x + y` performs simple addition on the two numbers `x` and `y`, even though both `x` and `y` are object types, and the plus operator is actually a function call to an overloaded operator routine. Though operator overloading is probably overused in general, there are many small programs that truly benefit from its existence.

Exceptions Once one has programmed with a strong exception model such as provided by C++ and Java, it is difficult to imagine any other reasonable model. The exceptions of the simulated program were implemented quite naturally with C++ exceptions.

Inlining Function inlining allows one to build zero-overhead abstractions. This was one of the goals of this project, and I relied heavily on inlining. Combined with overloaded operators, high level abstractions can be made quite inexpensive.

Good Compilers The MIPSpro C++ compiler I used generates excellent code and employs sophisticated optimization techniques. Such high-caliber compilers do not exist for most languages. Since one of the assumptions of the project was that I would rely on the compiler to optimize away the abstractions I introduced for the sake of readability, a good compiler was essential.

5.2 Hurtful Language Features

The design of the abstractions in the IRAM simulator were driven by a paranoia that the compiler would fail at any high-level optimizations. In order to maximize the odds that the compiler could optimize away all abstractions, the following features were avoided:

Dynamic Dispatch Only purely static data structures were used in the simulator. Any abstraction involving dynamic dispatch was avoided because truly heroic analysis is required to inline dynamically dispatched functions. Since I was attempting to build zero-overhead abstractions, function inlining was vitally important.

Pointers Pointer indirections were carefully avoided because they effectively kill all optimizations in the vicinity. Microbenchmarks of a simple 1-dimensional array abstraction revealed a 120x speedup when a seemingly innocent pointer indirection was removed. After this discovery, I removed all pointer indirections from the register file implementations in the simulator.

The result of avoiding dynamic dispatch and pointer indirection is a program where *all* routines can be statically inlined, and all data abstractions can be optimized completely away. In theory, there is nothing preventing the compiler from producing a simulator that would be competitive with the most highly-tuned C or even assembly language simulator.

5.3 Wished-For Language Features

An interesting approach to this project would have been to create a very high level language appropriate for this domain, and compile this language into C or C++. In the general case, this would have required building a compiler. However, some new features could be provided at a lower cost, by building a source-to-source translator. With the benefit of hindsight, this fantasy language would be most useful if it included the following features:

Safety It is generally accepted that high-level languages offer far more opportunity for optimization that low-level languages. Many of the difficulties in optimizing C and $\mathtt{C}^{++}$ programs stem from the fact that C is an unsafe language with pointers. Thus, much of the optimization effort is obfuscated by the presence of pointers, and much of the effort is spent optimizing pointers. There are many possible optimizations that our $\mathtt{C}^{++}$ compiler did not exploit that may be explained by problems inherent to the C programming language.

Dynamic Scoping It is clear that dynamic scoping is not a good replacement for static scoping. However, there are cases where dynamic scoping is extremely useful. For example, exception handlers have dynamic scope. In this project, there was a large emphasis on sharing code between different opcodes. This basically involves running the same code under the influence of different variable values. Modularity in this style requires either dynamic scoping, putting the code in a function and passing additional parameters, or global variables. The first option was not available, and the second could not be hidden well without a closure mechanism. For lack of an alternative, I settled on using global variables, but I suspect that using globals instead of dynamically scoped variables robbed the optimizer of important information. For example, a dynamically scoped variable could be declared constant, which would have enabled beneficial optimizations in our code. In addition, true modularity was not achieved because of the use of global variables.

User-Defined Operators In $\mathtt{C}^{++}$, most existing operators can be overloaded, but new operators cannot be defined. Defining new operators can certainly be abused, but in some cases it can be extremely useful. We could have used mathematical symbols instead of function calls in several high-visibility code sequences if we could have defined new operators.

Strong Exception Model The $\mathtt{C}^{++}$ language has built-in exceptions, but not as strongly as other languages such as Java. I mapped $\mathtt{C}^{++}$ exceptions to implement IRAM architectural exceptions in the simulator. A stronger exception model that allows static exception analysis by moving exceptions into the type-system would have helped.

Multiple Return Values This is perhaps the most unusual request, but would be simple to implement in a source-to-source translator. A common idiom in pseudo-code is for operations to return multiple results. This ability would have aided in producing the illusion that the instruction definitions were actually pseudo-code and not executable code. The CLU programming language has this feature.

New operators and multiple L-values could be provided relatively easily with a source-to-source translator. Safety, dynamic dispatch, and strong exceptions are deep properties of a language that cannot be provided if the target of the translator does not already support them. Since the target would have been C or $\mathtt{C}^{++}$, these features could only be provided at prohibitive costs. Using an existing language directly had many benefits, and, in retrospect, was the most effective use of my time. It is interesting, nevertheless, to imagine the most ideal language for this particular application.

6 Conclusion

This paper describes the implementation of the IRAM architectural manual and simulator, which are derived from a single parent document. The technology that enables this style of programming is called *literate programming*. The IRAM architecture manual is structured like a traditional architecture manual, yet the high-level code chunks that precisely define each instruction can be compiled into an architectural simulator. Not only is this compilation possible, but the simulator is very fast – fast enough to be the reference model for the IRAM project. By extracting the simulator from the architecture manual, debugging the simulator actually removes bugs from the manual, thereby improving the quality and consistency of the documentation.

Many simulator details are hidden from the reader of the architecture manual, and a conscious methodology was used to pick those details that should be obscured. All control flow issues and all exception conditions were exposed in the instruction definitions, while the arithmetic and logical operator details were hidden in utility routines and overloaded operators. This emphasis reflects the fact that control flow and exceptions are the complex and differentiating parts of most architectures. Finally, as is done in traditional architecture manuals, we were able to appeal to the context a reader provides to the instruction definitions, so that pedantic low-level details such as variable types and precise operator definitions could be omitted from the high-level code.

We measured the overhead incurred by writing very high-level $\texttt{C}^{++}$ code that uses many data abstractions and overloaded operators. By avoiding certain language features such as dynamic abstraction and pointer indirection, we were able to keep the overhead down to 20-30%. Though it seems to the author that the compiler should be able to reduce the overhead to zero, we were satisfied with the 10-13x improvement in execution time that compiler optimization provided. A 20-30% overhead remains small in face of the benefits of this approach. The main benefit is that we are able to verify the correctness of the architectural manual by comparing the architectural simulator to our RTL model. The ability to write verifiable documentation is the primary contribution of this research. The concept is proven that literate programming can remove a sometimes frustrating level of obscurity that separates traditional architecture manuals and software implementations.

References

[1] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[2] D. E. Knuth. The web system of structured documentation. Technical Report 980, Stanford University, 1983.

[3] D. E. Knuth. Literate programming. Technical report, Stanford University, 1992.

[4] Donald E. Knuth. *TEX and METAFONT: New Directions in Typesetting.* Digital Press, 1979.

[5] Donald E. Knuth. *The TeXbook.* Addison-Wesley, 1984.

[6] Donald E. Knuth. *TEX: The Program.* Addison-Wesley, 1986.

[7] David Martin. Vector extensions to the MIPS-IV instruction set architecture (the IRAM architecture manual). Technical report, UC Berkeley, TBD.

[8] Norman Ramsey. Literate programming simplified. *IEEE Software*, 1984.