Verischemelog: Verilog embedded in Scheme

James Jennings Eric Beuscher
Department of Electrical Engineering and Computer Science
Tulane University, New Orleans, Louisiana (LA) USA 70118
{jennings|beuscher}@eecs.tulane.edu

Abstract

`Verischemelog` *(pronounced with 5 syllables, ver-uh-scheme-uh-log) is a language and programming environment embedded in Scheme for designing digital electronic hardware systems and for controlling the simulation of these circuits. Simulation is performed by a separate program, often a commercial product.* `Verischemelog` *compiles to Verilog, an industry standard language accepted by several commercial and public domain simulators.*

Because many design elements are easily parameterized, design engineers currently write scripts which generate hardware description code in Verilog. These scripts work by textual substitution, and are typically ad-hoc and quite limited. Preprocessors for Verilog, on the other hand, are hampered by their macro-expansion languages, which support few data types and lack procedures. `Verischemelog` *obviates the need for scripts and preprocessors by providing a hardware description language with list-based syntax, and Scheme to manipulate it.*

An interactive development environment gives early and specific feedback about errors, and structured access to the compiler and run-time environment provide a high degree of reconfigurability and extensibility of `Verischemelog`.

1 Introduction

1.1 The Verilog Language

In this paper we describe a language for digital hardware design called `Verischemelog`, which compiles to Verilog.[1] Currently undergoing standardization, Verilog is a popular input language to sophisticated simulators of digital electronic circuits. Commercial simulators which accept Verilog input are used heavily in industry. Verilog is actually two languages: one describes hardware, the configuration of logic gates, wires, and other components; the other controls the event-based simulator, specifying input signals to circuits and the printing of textual output. The former part of Verilog is truly a configuration language, describing a static structure, and is sometimes called Verilog Hardware Description Language, or Verilog HDL. The other part of Verilog is sometimes called its "behavioral language," an apt name because it specifies the behavior of the simulation. Explicit in the behavioral language are the passage of time, the occurrence of events, and general computation as well. Figure 1 shows Verilog code for a half-adder, and behavioral code for testing it on one possible set of inputs. Verilog has a C-like syntax [KR88], both for hardware specification and for behavioral code. The semantics of the Verilog behavioral language are complicated by the need to explicitly allow simulation time to pass, and by the possibly ambiguous effects of statements which execute "at the same (simulation) time." In this paper we focus on hardware specification instead of simulation control for two reasons: the `Verischemelog` behavioral constructs simply mirror those of Verilog; and it is the need to *automatically synthesize* hardware description code which led us to develop `Verischemelog`.

1.2 Limitations of Verilog

The biggest limitation of the Verilog hardware description language is the lack of a facility for *generating* hardware description code. Verilog has a small macro language, essentially based on textual substitution, which does not allow, e.g. iteration. Therefore, although many designs are parameterized, there are no facilities for writing procedures which, when executed, *generate* hardware descriptions. Common design elements in digital systems include finite state machines (which are parameterized by their transition table),

[1] Verilog is a trademark of Cadence Design Systems, San Jose, CA, makers of the Verilog XL simulation system. Verilog and VHDL are popular standards.

```verilog
// Half adder with propagation delay 10

module half_adder (bit1, bit2, sum, carry);
input   bit1;
input   bit2;
output  sum;
output  carry;

and #10 anon1(carry, bit1, bit2);
xor #10 anon2(sum, bit1, bit2);

endmodule
```

(a) Verilog hardware description

```scheme
;; Half adder with propagation delay 10

(defmodule half_adder

  (interface (input bit1 bit2)
             (output sum carry))

  (description "Half adder with delay 10")

  (and (10) (carry bit1 bit2))
  (xor (10) (sum bit1 bit2)))
```

(b) `Verischemelog` *hardware description*

```verilog
// Tests half adder on 1 + 1

module test_half_adder;

reg     in1;
reg     in2;
wire    sum;
wire    cout;

half_adder      anon1(in1, in2, sum, cout);

initial
 begin
   in1 = 1;
   in2 = 1;
   #10;
   $display(
    "In: %d + %d ==> Sum: %d  Carry: %d",
     in1, in2, sum, cout
     );
 end
endmodule
```

(c) Verilog test program

```scheme
(defmodule test_half_adder
  (interface)
  (description "Tests half adder on 1 + 1")

  (reg in1 in2)
  (wire sum cout)
  (half_adder (in1 in2 sum cout))

  (initial
   (set! in1 1)
   (set! in2 1)
   (delay 10)
   ($display
     "In: %d + %d ==> Sum: %d  Carry: %d"
      in1 in2 sum cout)))
```

(d) `Verischemelog` *test program*

Figure 1: A half-adder computes the sum and carry of two 1-bit inputs. A Verilog module implementing a half-adder is shown on the left *(a)*. On the right *(b)* is the `Verischemelog` equivalent. (c) and (d) are test programs that direct the simulator to provide a high signal, 1, for each of the input signals and then to display the output signals after a delay of 10 simulation time units. When a hardware module is instantiated, the module name appears first (followed by a propagation delay for logic gates), then the name for this particular instantiation, and then a connection list. The latter is a list of wires which must match the module's interface. The Verilog convention for logic gates is that **the output signal is listed first**.
The Verilog `#`n syntax indicates a delay of n simulation time ticks, as do the `Verischemelog` `delay` form and the numeric parameter between the gate name and the connection list. Verilog requires instantiated modules to have individual names; in `Verischemelog` they are optional.

arithmetic operations (parameterized by the size in bits of operands and results), multiplexers (parameterized by the number of inputs and their size in bits), etc. Many web sites by and for designers who use Verilog promote the sharing of techniques for writing scripts which produce hardware description code.[2] One engineer writes, "I have written hundreds (well, maybe I exaggerate a bit) of Perl, awk, and cshell [sic] scripts for processing my verilog code and for synthesis" [unk98a].

We have examined many of the scripts available online. They vary widely in implementation language, programming style, and documentation. They have in common the ad-hoc nature of translators and compilers which were not written using traditional techniques such as lexical analysis, parsing, etc. To supplement the efforts of the script writers, a few programmers have contributed preprocessors which do operate on syntactic structures. These also vary widely, although one in particular, `vpp` [unk98b], has both a small sublanguage of C-like mathematical expressions and a small set of iteration constructs. These facilities notwithstanding, `vpp` is fundamentally limited by its translation language which lacks procedures and has only integer and floating point data types.

1.3 Design of `Verischemelog`

`Verischemelog` is a replacement for Verilog which is designed to alleviate the need for ad-hoc scripts and limited preprocessors. `Verischemelog` has the following properties:

- `Verischemelog` is embedded in Scheme [CR98] and has Scheme-like list-based syntax.
- Scheme is used as the "macro language" for generating hardware description and behavioral test code.
- Many errors are reported interactively, giving more specific and immediate feedback than, for example, the commercial Cadence Systems simulator.
- `Verischemelog` compiles to Verilog, an industry standard.
- `Verischemelog` easily interfaces to existing Verilog code.
- The programming environment includes project management facilities in Scheme.
- Quantitative descriptions of hardware modules, including gate count and maximum signal propagation delay, can be automatically computed. The calculation of these and other attributes can be programmed at the user level.
- The system is extensible by the user, who may add new operators or procedures to `Verischemelog`, or even reconfigure the code generator.

Languages like Perl are popular for writing scripts which generate Verilog hardware descriptions because they have a rich set of string processing functions. By using Scheme instead, and operating on list-based structures instead of strings, the `Verischemelog` user has an advantage over the Verilog user which is analogous to the advantage of Scheme/Lisp macros over those of C (as implemented by `cpp`). In other words, `Verischemelog` provides Scheme as a macro language. The transformation and generation of hardware description code in `Verischemelog` relies on the list-based syntax of `Verischemelog` and the facility with which Scheme manipulates list structures.

Below in Section 2 we illustrate some notable features of `Verischemelog` using brief examples, and in Section 3 we present some longer examples from two parameterized CPU designs. In Section 4 we show how the user can access the internal system to reconfigure the compiler and add custom project management tools. Section 5 discusses the implementation, and Section 6 summarizes some related work.

2 Using Verischemelog

`Verischemelog` was designed in part for instructional purposes, as an alternative to Verilog for a senior-level undergraduate course in Computer Architecture, and as an example of a domain-specific language with a compact and accessible implementation for students studying programming languages and compilers. The considerable efforts put into scripts for synthesizing Verilog code by practicing engineers indicates that it may be appropriate for industrial use as well. In either environment, academia or industry, we assume the user is familiar with the Verilog language, and many constructs in `Verischemelog` exploit that familiarity by simply mirroring Verilog constructs with Scheme-like syntax. We also assume that users will design large systems in a modular fashion, and that some

[2] A search of Yahoo (`yahoo.com`) gives 52 Verilog sites, many of which contain scripts, tips, and commercial design tools. AltaVista (`altavista.com`) reports 48740 sites referring to Verilog.

modules will be written in Verilog itself. Finally, although Verilog is for us the target language, we expect that `Verischemelog` output may at times be read by humans, for instance by another designer who does not use `Verischemelog`. For this reason we have provided a simple facility for commenting `Verischemelog` programs such that the comments are transferred to the output code upon compilation. The output code is also properly indented, for the same reason.

2.1 Key Concepts

We note that a *module* is a unit of code in Verilog which may contain hardware description, behavioral code, or a mixture of the two. Modules contain instantiations of other modules, with their dependencies forming a directed acyclic graph. For example, the half-adder modules in Figure 1 *(a)* and *(b)* each instantiate one `and` gate and one `xor` gate, both of which happen to be primitive modules. `Verischemelog` preserves the character of Verilog's modules, which are unrelated to any modules or packages one may find in Scheme.

Verilog hardware modules have *interfaces* which specify the input and output signals to the module, giving them internal names and types. An instantiation of a hardware module must provide wires of the proper size and type, connected in the proper order, which is specified by the module's interface. In this context, the size of a wire is the number of conductors it carries. For example, a small bus may carry 8 wires. Wire types include *input*, *output*, *ground*, and several others. Deciding whether two types match is usually straightforward.

The *simulator* is used to test hardware designs. It is not part of `Verischemelog`. The simulator is said to *execute* behavioral code which provides input signals to a hardware design and displays output such as the values of output (and internal) signals of the module. Naturally, the test program may change the input signals over time, and make choices about what to display and when. Verilog's behavioral language, in which test programs are written, is much like a general purpose procedural programming language in which computation consumes zero simulation time. `Verischemelog` provides list-based syntax for Verilog behavior constructs and checks for several categories of errors, but provides no new behavior language constructs.

2.2 Working Interactively

Development in Verilog follows the usual sequence of edit, compile, and execute.[3] In `Verischemelog` one works interactively at the Scheme read-eval-print loop. Using the Emacs editor with a `Verischemelog` session running in an editor buffer makes the experience quite similar to developing in Lisp or Scheme.[4] Of course, one is conscious of the differences in both striking and subtle ways. The `defmodule` form, which ultimately defines a Verilog module, and its attendant declarations of wires and gates clearly mark the domain. A more subtle difference between `Verischemelog` and Scheme appears in the form of restrictions on symbol names in `Verischemelog`, which conform to Verilog's C-like syntax, containing only the underscore character and alphanumerics. We chose to use this syntax rather than translate from the larger universe of Scheme symbols in order to allow humans and programs to easily match `Verischemelog` (source language) names with their Verilog (target language) counterparts, e.g. in messages produced by the simulator during execution.

Hardware development then follows this process:

1. `Verischemelog` hardware description and test code is developed interactively using the `defmodule` form which invokes the front-end of the `Verischemelog` compiler, but does not generate Verilog code. It is here that syntactic and other errors are reported, and interfaces between modules are checked for consistency.

2. When a module is to be tested using the simulator, it is compiled using the `compile` procedure which generates Verilog code for the target module as well as conditionally for any modules on which it depends.

3. The simulator is invoked on the generated Verilog code, using a script written by `Verischemelog` which enumerates the output files needed and contains any desired simulator options. In a Scheme which supports such operations, `Verischemelog` can run the simulator in batch mode in a child process, displaying its output, etc.

In the sections that follow we show how `Verischemelog` can be used to automatically synthe-

[3] Some simulators, like Cadence Systems Verilog XL, combine compilation and subsequent execution into one step by default.

[4] We do not address in this paper the possible "culture gap" which in theory might face engineers trained in C-like languages who are presented with `Verischemelog`.

size hardware description and behavioral test code. We begin with the most basic feature, escaping to Scheme.

2.3 Escaping to Scheme

First note that we distinguish *evaluation* of a `defmodule` form from *compilation*. To evaluate is to run the front end of the `Verischemelog` compiler, including the type checker; to compile is to perform code generation. The `defmodule` form is implemented as a Scheme macro which evaluates Scheme code embedded in the `defmodule` body, inlines the results, and finally evaluates the module definition. Embedded Scheme forms begin with one of the reserved keywords `scheme` or `scheme-splicing`, which can be abbreviated using `$` or `$$` respectively.

Both `scheme` and `scheme-splicing` behave as a Scheme `begin` form: each subform is evaluated in sequence, and the value of the last form is returned. These forms parallel Scheme's `unquote` and `unquote-splicing`. The value of a `scheme` form is inserted in place directly into the surrounding `defmodule` code. The value of a `scheme-splicing` form must be a list; the list is "spliced into" the surrounding code.

Both escape forms may be used to generate hardware description code. A more basic use is to retrieve a value. For example, the propagation delays for the gates in the half-adder of Figure 1 could be retrieved from the Scheme variable `*delay*` as follows:

```
(define *delay* 14)

(defmodule half_adder

  (interface (input bit1 bit2)
             (output sum carry))

  (description
   (string-append "Half adder with delay "
                  (number->string *delay*)))

  (and ((scheme *delay*)) (carry bit1 bit2))
  (xor ((scheme *delay*)) (sum bit1 bit2)))
```

Note that the syntax for gate instantiations requires the delay to be in parentheses. Therefore:

`((scheme *delay*))` $\Rightarrow$ `(14)`.

The same parameter could be used in the corresponding behavioral program which tests this half-adder, thus ensuring their consistency.

Note also that `Verischemelog` evaluates the expression in the `description` form, which must return a string. The `description` form (and a similar `comment` form) exist only in `Verischemelog`, not Verilog, and so the automatic evaluation without use of an escape form is mellifluous.

2.4 Synthesizing Systems

A common situation calling for automatic generation of hardware description is that of parameterizing a design element, such as an adder, by the number of bits in the word size of a Central Processing Unit (CPU). In this case we are synthesizing an entire system (a CPU) which may have many variable parameters, one of which, the word size, parameterizes the adder that will be used in the Arithmetic and Logic Unit (ALU). We may proceed in one of two ways. We may simply use an escape form such as `scheme-splicing` to generate a series of full-adders (Figure 2) or we may define a Scheme procedure which contains a `defmodule` form. For illustration, we will show the latter, the code for which is in Figure 2. Figure 3 shows the Verilog code resulting from `(make-adder 3)`.

These very small examples serve as useful illustrations, but the true utility of `Verischemelog` can only be seen when designing large systems, such as a CPU. A CPU may be parameterized by word size, instruction format, bus configuration, etc. During the design process, a component such as the ALU may be implemented in a general form, parameterized like the adder of Figure 2. (This may encourage code reuse, as alternate configurations are easily generated from one implementation.) Other components, such a microprogrammed control unit, may be designed in a fixed (not parameterized) form.

Even in a fixed form, however, a control unit is easier to generate in `Verischemelog` than in Verilog, because attributes can be represented symbolically in Scheme data structures. For example, a Scheme variable can hold a list of symbols representing all of the control signals in the CPU. The control unit module can be built separately from the CPU, and we can ensure their interfaces will match because `Verischemelog` can *generate* their interfaces using the control signal list. Also, control signals can then be referred to symbolically, by their names, with `Verischemelog` automatically mapping the names to indexed references into a control bus such as `controls[27]`.

Likewise, in a few lines of Scheme one can write a small assembler for generating memory images to

```
(define make-adder
  (lambda (wordsize)
    (let ((desc
           (string-append (number->string wordsize)
                          "-bit adder with carry out"))
          (full-adder
           (lambda (carryin a b sum carryout)
             `(full_adder (,carryin ,a ,b ,sum ,carryout)))))

      (defmodule adder
        (description desc)
        (interface (input ((scheme wordsize)) a b)
                   (output ((scheme wordsize)) sum)
                   (output carry))

        (supply0 ground)
        (wire ((scheme (- wordsize 1))) c)

        (scheme-splicing
         (iterate wordsize
          (lambda (i)
            (full-adder (if (zero? i)             ; carry in:
                            'ground               ;          0,
                            (: 'c (- i 1)))       ;          or c[i-1]
                        (: 'a i)                  ; input a[i]
                        (: 'b i)                  ; input b[i]
                        (: 'sum i)                ; sum[i]
                        (if (= i (- wordsize 1))  ; carry out:
                            'carry                ;          module output,
                            (: 'c i))))           ;          or c[i]
          ))))))
```

```
(defmodule full_adder                   ; two half-adders make a full-adder.
  (interface (input carry_in bit1 bit2)
             (output sum carry_out))
  (wire temp_sum temp_carry1 temp_carry2)
  (half_adder half_1 (bit1 bit2 temp_sum temp_carry1)
              half_2 (carry_in temp_sum sum temp_carry2))
  (or (10) carry_or (carry_out temp_carry2 temp_carry1)))
```

Figure 2: A Scheme procedure which generates `Verischemelog` code for a "ripple" adder which contains *wordsize* full-adders. The input signals to full-adder i are bit i of each input wire a and b, and the carry out wire of the previous adder, $c[i-1]$. The carry in of the first full-adder is wired to 0 (ground), and the carry out of the last full-adder is an output wire of the adder module.
The `:` procedure takes a symbol (the name of a wire) and one or two numbers (indices) and generates an array-like reference. Multiple-conductor wires in Verilog are declared as arrays with their leftmost and rightmost bit indices given, as in `input [2:0] a`, which declares a three-conductor wire named `a` whose conductors are numbered from left to right: 2, 1, 0. The procedure `iterate` is like a procedural form of Lisp's `dotimes`, e.g.

`(iterate 5 (lambda (x) x))` ⇒ `'(0 1 2 3 4)`.

```
// 3-bit adder with carry out

module adder (a, b, sum, carry);

input   [2:0] a;
input   [2:0] b;
output  [2:0] sum;
output  carry;

supply0 ground;
wire    [1:0] c;

full_adder anon1(ground, a[0], b[0], sum[0], c[0]);
full_adder anon2(c[0], a[1], b[1], sum[1], c[1]);
full_adder anon3(c[1], a[2], b[2], sum[2], carry);

endmodule
```

Figure 3: The Verilog output from `(make-adder 3)`.

be used in testing CPU designs. In Section 3 below we describe two large projects, each a CPU design, in which these techniques were employed.

2.5 Evaluation Catches Errors

As mentioned above, `defmodule` forms are evaluated interactively. `Verischemelog` reports an error when any of the following occur:

- Syntax errors.
- Module interface does not match connection list (a form of type checking).
- The identifier `z` is used as a variable name.

Verilog allows the use of `z` as a variable, but it is also the name of a constant meaning "high impedance." Consequently, `z` is a "write only" variable in Verilog because when it appears in an expression it refers to the constant value. The lack of a warning or error by Verilog trips up many novice users, and probably others as well.

`Verischemelog` reports several types of warnings:

- Wire used but not declared.
- Module is missing a description, has a null body, is instantiated but not (yet) defined, etc.
- Signal possibly used as feedback.

These warnings illustrate how a customizable development environment can be used to promote good design practices. For example, Verilog allows the use of undeclared wires, but `Verischemelog` generates a warning because often this is the result of a typographical error. Similarly, `Verischemelog` encourages the use of the `description` form, which generates a block comment in the header of the output file. Descriptions of defined modules can also be searched interactively, to aid in project management. Finally, sometimes the output signal of a module is used inside the module itself as the input to another device. Although it can be done intentionally, some of the time this is the result of mis-wiring. The warning helps detect those cases, and it can be suppressed when the use of feedback is intentional.

2.6 Precompiled Modules

`Verischemelog` allows designers to declare modules "precompiled." The designer provides the module name, its interface, and the filename of the Verilog code which defines it. Although `Verischemelog` will compile code which instantiates unknown modules (and will warn about it), the ability to declare precompiled modules enables the same interface type checking that would occur if the instantiated modules had

```
(define (make-mux-n-to-1 n bits)
  "BITS is the word size of mux input, N is the max number of inputs"
  (let* ((name (lambda () (make-symbol 'mux_ n 'to 1 '_ bits 'bit)))
         (log-n (ceil-log-n n 2))
         (expt-n (expt 2 log-n)))
    (make-decoder-m-control-bits log-n)
    (make-tri-state-n-bits bits)
    (defmodule (name)
      (description desc)
      (interface
       (output ((scheme bits)) result)
       (input ((scheme bits))
              (scheme-splicing (map-bits n (lambda (n) (make-symbol 'in n)))))
       (input ((scheme log-n)) control))
      (wire ((scheme expt-n)) ndecode_out)

      ($$ (cons (list (make-symbol 'decoder_ log-n 'to expt-n)
                      (make-symbol 'd log-n 't expt-n)
                      (list 'ndecode_out 'control))
                (map-bits n
                          (lambda (n)
                            (list (make-symbol 'tri_state_ bits '_bits)
                                  (make-symbol 'ts bits '_ n)
                                  (list 'result
                                        (make-symbol 'in n)
                                        (: 'ndecode_out n))))))))))
(a).
```

```
module mux_2to1_4bit (result, in0, in1, control);

output  [3:0] result;
input   [3:0] in0;
input   [3:0] in1;
input   control;

wire    [1:0] ndecode_out;

decoder_1to2    d1t2(ndecode_out, control);

tri_state_4_bits        ts4_0(result, in0, ndecode_out[0]);
tri_state_4_bits        ts4_1(result, in1, ndecode_out[1]);

// delay of 16

endmodule

(b).
```

```
module mux_4to1_9bit (result, in0, in1, in2, in3, control);

output  [8:0] result;
input   [8:0] in0;
input   [8:0] in1;
input   [8:0] in2;
input   [8:0] in3;
input   [1:0] control;

wire    [3:0] ndecode_out;

decoder_2to4    d2t4(ndecode_out, control);

tri_state_9_bits        ts9_0(result, in0, ndecode_out[0]);
tri_state_9_bits        ts9_1(result, in1, ndecode_out[1]);
tri_state_9_bits        ts9_2(result, in2, ndecode_out[2]);
tri_state_9_bits        ts9_3(result, in3, ndecode_out[3]);

// delay of 24

endmodule

(c).
```

```
module mux_16to1_7bit (result, in0, in1, in2, in3, in4, in5, \
 in6, in7, in8, in9, in10, in11, in12, in13, in14, in15, control);

output  [6:0] result;
input   [6:0] in0;
input   [6:0] in1;
input   [6:0] in2;
input   [6:0] in3;
input   [6:0] in4;
input   [6:0] in5;
input   [6:0] in6;
input   [6:0] in7;
input   [6:0] in8;
input   [6:0] in9;
input   [6:0] in10;
input   [6:0] in11;
input   [6:0] in12;
input   [6:0] in13;
input   [6:0] in14;
input   [6:0] in15;
input   [3:0] control;

wire    [15:0] ndecode_out;

decoder_4to16   d4t16(ndecode_out, control);

tri_state_7_bits        ts7_0(result, in0, ndecode_out[0]);
tri_state_7_bits        ts7_1(result, in1, ndecode_out[1]);
tri_state_7_bits        ts7_2(result, in2, ndecode_out[2]);
tri_state_7_bits        ts7_3(result, in3, ndecode_out[3]);
tri_state_7_bits        ts7_4(result, in4, ndecode_out[4]);
tri_state_7_bits        ts7_5(result, in5, ndecode_out[5]);
tri_state_7_bits        ts7_6(result, in6, ndecode_out[6]);
tri_state_7_bits        ts7_7(result, in7, ndecode_out[7]);
tri_state_7_bits        ts7_8(result, in8, ndecode_out[8]);
tri_state_7_bits        ts7_9(result, in9, ndecode_out[9]);
tri_state_7_bits        ts7_10(result, in10, ndecode_out[10]);
tri_state_7_bits        ts7_11(result, in11, ndecode_out[11]);
tri_state_7_bits        ts7_12(result, in12, ndecode_out[12]);
tri_state_7_bits        ts7_13(result, in13, ndecode_out[13]);
tri_state_7_bits        ts7_14(result, in14, ndecode_out[14]);
tri_state_7_bits        ts7_15(result, in15, ndecode_out[15]);

// delay of 72

endmodule

(d).
```

Figure 4: (a) shows a mux-generator written in `Verischemelog`. (b), (c), and (d) show examples of muxes generated by the mux-generator with the following arguments: `(make-mux-n-to-1 2 4)`, `(make-mux-n-to-1 4 9)`, `(make-mux-n-to-1 16 7)`.

been defined in `Verischemelog`. The `defcompiled` form used to declare precompiled modules can even be generated automatically from Verilog source files.

Another important use of `Verischemelog` is for efficiency. When a project is large, modules not actively being modified can be declared precompiled so that only their interfaces are loaded into `Verischemelog`, saving memory and time. Naturally, `Verischemelog` can generate the appropriate `defcompiled` form for any module already defined, so that in subsequent sessions only the short `defcompiled` forms need to be loaded, instead of the much longer `defmodule` forms.

3 Examples

Tables 1 and 2 show the sizes of two hardware designs implemented in `Verischemelog`. Each is a complete CPU with memory (or memories) and a microprogrammed control unit. Each was designed from the outset to be scalable; consequently the same `Verischemelog` code was used to generate variations of each machine with different word sizes. To generate the machines shown in the tables, a single parameter (the data size) was changed, and the modules recompiled. To test each variation, a small assembler (included in the figures in the table) was used to produce a memory image (or images) containing a program and data. Also counted in the figures in the table are library procedures of general utility, such as `iterate`.

Table 1 reflects a silicon approximation to a Turing Machine, a small processor with separate data and instruction memories. The machine has four instructions, *right*, *left*, *jump*, and *halt*. The *jump* instruction performs an unconditional branch to an absolute address. The *right* and *left* instructions each take three operands: *match*, *replacement*, and *goto*. A special purpose register, H, contains a data memory address, initially 0. The instruction

```
right   a, b, l
```

does nothing when the contents of data memory at address H does not match `a`. The next instruction is fetched. If `a` is in memory at H, however, this instruction writes `b` there, increments H, and branches to `l`. The operation of `left` is analogous, but H is decremented.

The CPU of Table 2 has a more traditional design, with 16 general-purpose registers, an ALU supporting integer arithmetic and bitwise logical operations, three internal busses, and a RISC-style instruction set in which the ALU operates on data in registers and writes its result to a register. The instruction set contains about two dozen instructions, and the control unit is microprogrammed.

The tables illustrate that these parameterized designs required a substantial amount of `Verischemelog` code, but an amount roughly of the same order of magnitude as the generated Verilog code which otherwise might have been written by hand. For the Turing Machine, a 7 bit data size lets us write programs which manipulate ASCII characters on the tape, and we have not compiled a larger model. The processor of Table 2, on the other hand, has been compiled and tested for the purposes of illustration on data sizes so large as to be (currently!) impractical to build in silicon. We consider this a reasonable test of the HDL synthesis capabilities of `Verischemelog`.

4 Extending Verischemelog

An important aspect of `Verischemelog` is the extent to which it be customized, reconfigured, and extended by the user. Some of the customizations are:

- Selective control over the display of warnings
- Setting of various working and output directories
- Definition of simulator options
- Block comments and headers for output files

An important reconfiguration feature consists of a set of customizable tables used by the code generator. For example, users can interactively modify a list of known Verilog behavioral procedures. New built-in procedures in a new release of the simulator can be added this way. Users can also tell `Verischemelog` about new unary and binary operators. Finally, `Verischemelog` maintains a table of translations of operators and procedures from `Verischemelog` to Verilog. This allows `Verischemelog` users to write `=` instead of `==`, `bitwise-not` instead of `~`, etc.

One extensibility feature is the `verbatim` form, which allows the user to insert arbitrary strings into the Verilog output. While clearly of limited utility, it does provide a method of writing user-level code which generates Verilog output. Of course, such code is actually generated at evaluation time rather than code generation time. A structured way to change the code generator, for example to add new language constructs, is under consideration.

With respect to hardware analysis and project management, however, `Verischemelog` provides sufficient

Table 1: Various measures of the size of the Turing Machine implementation. Data size refers to the number of bits per cell on the tape, which was implemented as a finite memory. Measurements of lines of code do not include comments or blank lines. The term *definitions* refers to top-level `define` and `defmodule` forms. HDL code includes all forms necessary to generate Verilog HDL, including Scheme procedures. Test code includes all behavioral programs for testing the system and individual components.

Data Size	`Verischemelog` source		Verilog output
	Lines of code	Definitions	Lines of code
1 bit	732 HDL, 398 test	54 HDL, 35 test	530
2 bits	(same)	(same)	540
7 bits	(same)	(same)	589

Table 2: Various measures of the size of the EBP CPU implementation. Data size refers to the number of bits in a machine word (the size of the data bus, registers, ALU operands, etc.). Address size in each case was 16 bits. Measurements of lines of code do not include comments or blank lines. The term *definitions* refers to top-level `define` and `defmodule` forms. HDL code includes all forms necessary to generate Verilog HDL, including Scheme procedures. Test code includes all behavioral programs for testing the system and individual components. pThe numbers for the HDL code includes such high level abstractions as automatic generation of the control unit's lookup table and micromemory based on a symbolic description of the control signals, an assembler for writing memory images, and calculations of simulated hardware delays for choosing an optimal clock speed.

Data Size	`Verischemelog` source		Verilog output
	Lines of code	Definitions	Lines of code
16 bits	3246 HDL, 964 test	240 HDL, 27 test	1397
32 bits			1853
64 bits			2549
128 bits			4085
256 bits			7229
512 bits			13373
Totals	3246 HDL, 964 test	240 HDL, 27 test	30486 lines, 6 processors

structured access to its internals for the user to easily write code to:

- Generate a different script for invoking the simulator.
- Generate representations of module dependencies.
- Calculate the number of primitive gates or particular user-defined modules instantiated in a given set of modules.
- Calculate arbitrary statistics on hardware descriptions.

5 Implementation Notes

`Verischemelog` was written in Kali Scheme [CJK95], a variant of Scheme48 [KR94]. Scheme48 is a compact and portable implementation of Scheme with a module system, record package, and hygienic macros. Scheme48's Unix interface was sufficient to allow `Verischemelog` to launch the Verilog XL simulator in another process, and to display its output. The source code for the `Verischemelog` run-time environment, compiler, all data structures, and an online help facility totals 247 forms written in 3282 lines.

Kali Scheme is a novel distributed implementation of Scheme. By using it to implement `Verischemelog`, distributed synthesis, compilation, or analysis of large systems is possible. Owing to the strength of Kali's design, these distributed extensions can easily be written entirely in user space, and quite simply as well.

6 Related Work

The original inspiration for `Verischemelog` was a programming language and environment named THEE [Woo93]. THEE allows Common Lisp programmers to generate C code in much the same way that `Verischemelog` allows designers to generate Verilog.

Evidence that Scheme might mix well with a traditionally low-level "down and dirty" task such as hardware design came in the form of the Envision system [SF97] which extends Scheme for computer vision. The Envision environment contains a new language, statically-typed and with different semantics from Scheme in other respects, but with Scheme's syntax. Programmers write Scheme programs with embedded calls to procedures in the new language. Those procedures are transmitted to another process, the "co-processor," which interprets them. The embedded language, designed for image processing, is interpreted only during development. At any time code in the embedded language can be compiled to C and linked with the "co-processor" for subsequent high-performance execution when called from Scheme.

Finally, the structured access to the internals of `Verischemelog` was inspired by the notion of meta-object protocols [KdRB91].

7 Conclusion

With Scheme as a macro language, `Verischemelog` users can easily generate code which compiles to Verilog, making synthesis of digital hardware designs much more efficient than programming directly in the output language. By providing structured access to the compiler and run-time system, `Verischemelog` enables a great deal of customization, reconfiguration, and extension. By performing syntax and type checking interactively, users get specific and timely feedback about errors. A variety of warnings reinforce good coding style as well as point out possible problems. The ability to interface with existing Verilog code and to handle moderately large designs indicates that `Verischemelog` may be a practical tool for use in industry.

The most important aspect of `Verischemelog`, however, is the central idea that to allow users to effectively generate code, they should be given a language designed to manipulate that code. We used Scheme for that language, and designed the list-based syntax of `Verischemelog` to be manipulated by it.

Acknowledgments

The source code, grammar, examples (including the Turing Machine) and design documents of `Verischemelog` are available on the web from `http://www.eecs.tulane.edu/www/Jennings`.

This paper describes research done in the Department of Electrical Engineering and Computer Science at Tulane University.

References

[CJK95] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, September 1995.

[CR98] W. Clinger and J. Rees. Revised^5 report on the algorithmic language scheme. *ACM Sigplan Notices*, 33(9):26–76, September 1998.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KR88] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[KR94] Richard Kelsey and Jonathan Rees. A tractable scheme implementation. *Journal of Lisp and Symbolic Computation*, 7:315–335, 1994.

[SF97] Daniel E. Stevenson and Margaret M. Fleck. Programming language support for digitized images or, the monsters in the closet. In *Usenix conference on Domain-Specific Languages*, pages 271–284, 1997.

[unk98a] Author unknown. Celia's verilog page. `http://www.teleport.com/~celiac/tools.html`, last verified Sat Oct 3 13:03:13 CDT 1998.

[unk98b] Author unknown. Vpp, a verilog preprocessor. `http://www.sybarus.com/product.htm`, last verified Thu Oct 8 17:03:13 CDT 1998.

[Woo93] John Woodfill. The programming environment and language thee (tm). *Unpublished source code*, ©1990, 1991, 1992, 1993.