

esim: A Structural Design Language and Simulator for Computer Architecture Education

Ethan Miller^{1,2}
elm@acm.org

Jon Squire¹
squire@csee.umbc.edu

¹ Computer Science & Electrical Engineering Dept., University of Maryland, Baltimore County

²Department of Computer Science, University of California, Santa Cruz

ABSTRACT

We have developed a structural design language for use in undergraduate computer architecture classes that has much of the power of VHDL with little of the complexity. This language, *esim*, allows students to build arbitrarily complex digital logic designs using simple hierarchical design techniques. Students can simulate and debug their designs using a simulator implemented as a Tcl module. Because *esim* was not intended for designing and building physical circuits, it omits many of the primitives necessary for “real” hardware design and instead focuses on the concepts necessary for teaching students about digital designs.

We have used *esim* as a teaching tool in undergraduate computer architecture classes at UMBC for several semesters. Students in these classes have implemented projects as complex as a pipelined RISC processor and a full 16x16 combinational multiplier. The compiler and simulator for the language are freely distributable, and may be expanded using standard Tcl packages and Tcl code. Current simulator modules include one for displaying signal values on the screen; modules that graph signal values are planned.

1 Introduction

Computer architecture has traditionally been a relatively difficult subject for both students and instructors because of the difficulty in providing good tools for “hands-on” learning. Although most computer science students are familiar with programming languages such as C and Java, they have little experience with hardware design languages.

A good hardware design language is a very useful tool for teaching computer architecture, but common hardware design languages such as VHDL [1,2] and Verilog [5] suffer from several problems. First, both languages are very complex because they need to be usable for real hardware design. This complexity makes it more difficult for undergraduates, many of whom have little interest in architecture beyond taking an introductory course, to learn the language. Second, the tools necessary to run both languages are both complex, expensive, and resource-intensive. While this may be an acceptable tradeoff for stu-

dents in an advanced digital design class, it is undesirable for a core class required of all computer science majors. The alternative approach of using graphical design tools to wire components together has a similar problem — existing tools are either too powerful and complex (and expensive) or too simple for use in a computer architecture class.

We have developed a new hardware design language, *esim*, for use in computer science classes. Because we designed the language for instructional rather than production use, we were able to leave out many of the more complex constructs present in VHDL and Verilog. The language is purely digital — signals can only take values of **0**, **1**, **conflict**, and **undriven** — and has primitives for both regular signals and memory, providing an efficient way to simulate microprocessor designs with registers and caches. *Esim* encourages hierarchical design by allowing the inclusion of modules in other modules.

Esim designs can be simulated easily and quickly on an inexpensive PC running Linux or other Unix-like operating system. The simulator is implemented as a Tcl [6] module, so it can be programmed to provide inputs to particular signals in the circuit. In addition, Tcl has facilities for graphical display, allowing the development of graphical signal displays. We have used *esim* for several semesters in a junior/senior-level computer architecture class, and feedback from students has been quite positive.

This paper describes the *esim* hardware design language, and includes short samples of designs written in *esim*. We will not include a formal description of the *esim* language; rather, we will informally describe the language and how to use it to implement computer architecture class assignments. We will then describe the Tcl interface to *esim*, which allows students and instructors to write scripts to control their simulations. Tcl also allows the construction of display tools that can help students understand what is happening in the projects they have designed. The *esim* language described in this paper is freely available at <http://www.csee.umbc.edu/~elm/esim>; installation requires only standard GNU/Unix tools such including `gcc`, `flex`, `bison`, and `Tcl/Tk`.

2 Hardware descriptions in *esim*

Esim is a simple hardware design language that allows the implementation of large, complex digital design projects using hierarchical design techniques. The language corresponds closely to hardware components such as AND and OR gates, but allows users to build up complex designs from simple reusable components. As with most hardware design languages, all operations occur in parallel; *esim* is not a sequential language. This difference from software programming languages is the largest conceptual problem for computer science students who are used to programs that execute one line at a time.

2.1 Data types

Esim has two data types: *signal* and *memory*. A single signal may have one of four different values: 0, 1, Z (undriven) or X (conflict/indeterminate). Sig-

nals may be aggregated together; for example, the definition

```
signal inputA[32];
```

declares a signal *inputA* that is 32 bits wide. Any value that is assigned to *inputA* must then be exactly 32 bits wide, and may include any combination of 0, 1, Z, and X values. A definition may also include an initial value for a signal specified as a sequence of bits. For example,

```
signal inputA[32] <= #h01234567;
```

states that *inputA* should be given the initial value 0x1234567. This is most useful for initializing signals such as clocks that oscillate between two values; if no initial value is given, the signal will be stuck at X for the entire simulation.

Values in *esim* may be specified as either hexadecimal or binary using the `#h` and `#b` notations, respectively. Values are not case-sensitive (*a* and *A* both refer to the hex digit whose value is 10), and may include undefined (X) and undriven (Z) digits. For hexadecimal numbers, X and Z digits translate four consecutive X or Z digits, respectively. Thus, the hex value `#h9X3Z` would be equivalent to the binary value `#b1001XXXX0011ZZZZ`.

The *memory* data type is used for structures such as register files or main memory. It is implemented very efficiently; memories of megabytes can be simulated on inexpensive PCs. A memory is defined as:

```
memory reg[1024];
```

Values in a memory remain until they are changed by a memory statement (see Section 2.2). Memory cannot be initialized from within an *esim* program, but the simulator can read and write memory values. By using Tcl scripts and memory write commands, a memory can be loaded with the appropriate values.

2.2 Assignments

All statements in *esim* operate in parallel; essentially, each statement defines a gate or small collection of gates. A single statement in *esim* is of the form

```
signal <= expression;
```

where *expression* may include signals and other expressions combined by the operators listed in Figure 1. The only restriction on composing expres-

<code>a & b</code>	<code>a</code> AND <code>b</code>
<code>a b</code>	<code>a</code> OR <code>b</code>
<code>a ^ b</code>	<code>a</code> XOR <code>b</code>
<code>~a</code>	NOT <code>a</code>
<code>a[x:y]</code>	bits <code>x</code> down to <code>y</code> of signal <code>a</code> (result is <code>x-y+1</code> bits wide)
<code>a == b</code>	<code>a</code> is equal to <code>b</code> (result is 1 bit wide)
<code>a != b</code>	<code>a</code> is unequal to <code>b</code> (result is 1 bit wide)
<code>a . b</code>	<code>a</code> concatenated with <code>b</code> (result is as wide as <code>a</code> and <code>b</code> combined)
<code>a when cond else b</code>	<code>a</code> whenever <code>cond</code> (a 1 bit wide signal) is 1, <code>b</code> otherwise
<code>#h0123</code>	constant in hexadecimal (16 bits wide in this example)
<code>#b0101</code>	constant in binary (4 bits wide in this example)

Figure 1. Operators in *esim*.

```
reg read s from rs . #b00000 when #b1; // always enabled
reg write d to rd . #b00000 when wEnb on rising clk;
```

Figure 2. Sample memory statements.

sions is that the number of bits for a two-input operator must match.

All of the binary operators (AND, OR, XOR) are bitwise operators, as are tests for equality; if the inputs to an operator are wider than 1 bit, the operation is applied bit by bit and placed into the corresponding position in the result. This restriction applies to constants as well; `#h012` is a 12-bit wide signal.

Most operators produce a result as wide as that of either input. However, some operators produce a result of different width. For example, equality and inequality operators produce a one-bit result regardless of the width of their inputs.

In most expressions, constants have the interpretation described in Section 2.1. However, comparison operations (equality and inequality) use a slightly different definition. For these operators, a constant of `X` means “don’t care,” and will match *any* input signal (including `X` and `Z`). There is thus no easy way to construct a test that will be true *only* if a signal is undetermined. For example, the expression `a==#h8X` will result in 1 as long as the high-order 4 bits of `a` are **1000**.

Another statement type involves memories, which can be both read and written. Sample memory state-

ments are shown in Figure 2. The first statement reads a value from memory into `s` using the address (`rs . #b00000`) and reading as many consecutive bits from this address as `s` is wide. Since the read statement is always enabled, values are always put into `s`. If the memory were not enabled, it would output `Z` values. The second statement writes a value into the memory when `wEnb` is set to 1. However, values are only written when `clk` rises from 0 to 1.

Assignments in *esim* can take two types of modifiers. The first type of modifier is the *after* modifier, indicating that the assignment takes place after the specified delay. For example

```
a <= b after 10ns
```

means that `a` will get `b`’s value after 10 ns of simulated time have passed. This modifier can be used to simulate gate delays; the default delay for *esim* assignments is 5 ns. The second type of modifier is the *on* modifier. This modifier can be of the form *on rising a* or *on falling b*. In the first case, the assignment in the modified statement only takes place when `a` changes from 0 to 1. In the second case, the assignment only occurs when `b` changes from 1 to 0. These two modifiers can be combined to result in a delayed assignment when a clock signal rises or falls.

```

define regfile (rd[3], rs[3], d[16], s[16], wEnb, clk)
memory reg[128];
circuits
  reg read s from rs . #b0000 when #b1; // always enabled
  reg write d to rd . #b0000 when wEnb on rising clk;
end circuits;
end regfile;

```

Figure 3. Definition of a register file in *esim*.

```

// 8-bit latch clocked on the clk signal when enb1 and enb2 are both enabled
define latch8 (q[8], d[8], enb1, enb2, clk)
signal enabled;
signal qInternal[8];
circuits
  qInternal <= d when (enb1 & enb2) else qInternal;
  q <= qInternal on rising clk;
end circuits;
end latch8;

// Set up a 16 bit latch as 2 8-bit latches
define latch16 (q[16], d[16], clk)
circuits
  low use latch8 (q[7:0], d[7:0], #b1, #b1, clk); // always enabled
  high use latch8 (q[15:8], d[15:8], #b1, #b1, clk); // always enabled
end circuits;
end latch16;

signal q[32];
signal d[32];
circuits
  low use latch16 (q[15:0], d[15:0], clk);
  high use latch16 (q[31:16], d[31:16], clk);
end circuits;

```

Figure 4. Full implementation of a 32-bit latch in *esim*.

2.3 Components

Basic hardware modules defined in *esim* are called *components*. A sample component (a register file with 8 registers of 16 bits each) is shown in Figure 3. Parameters to the component definition do not specify input or output; rather, parameters are simply placeholders allowing external signals to be used by the internals of the component. Parameters may be either a single bit wide (*wEnb*, *clk*), or multiple bits wide (*s* is 16 bits wide). Internal signals and definitions are not visible outside the component, though their effects may be. For example, components that include *regfile* may not directly reference *reg*, though they can store and read values by appropri-

ately manipulating the addresses and control signals of *regfile*.

Components may themselves include other components, as shown in Figure 4. This is done via the *use* statement, which assigns different names to the different instances of each component. In the Figure 4 example, the first instance of *latch8* is named *low*, and the second instance is named *high*. Note that parameters to an instance can include signals, signal slices, and constants.

```

define foo (a, b)                // define component foo
circuits
    a <= b & 1;
end circuits;
end foo;

define bar (x[2], y[2])          // define component bar
circuits
    s use foo (x[0], y[0]);      // use component foo in bar
    t use foo (x[1], y[1]);
end circuits;
end bar;

signal topx[4], topy[4];        // main circuit
circuits
    hi use bar(topx[3:2], topy[3:2]); // use component bar in main
    lo use bar(topx[1:0], topy[1:0]);
end circuits;

```

Figure 5. Sample *esim* code demonstrating component and signal naming.

2.4 Building a full hardware description

Figure 4 contains a complete description of a 32-bit latch. It includes two components, *latch8* and *latch16*, and a top-level circuit that has two instances of *latch16*. The signals in the top-level circuit are passed down to the *latch16* and *latch8* instances as needed.

Esim is compiled in two passes. In the first pass, the input file is parsed and converted to an internal representation. The second pass creates an output file, similar to a netlist, that is used as input to the simulator. Because the input file is parsed in a single pass, forward references are not allowed — every component must be defined before it used. However, there is no limit to nesting level; *esim* allows as many levels of hierarchy as necessary. If the compilation is successful, a netlist file suitable for input to the simulator is generated.

3 Using the *esim* simulator

The *esim* simulator is embedded within a Tcl shell, and can be programmed and operated just as any other Tcl/Tk program. It can take advantage of Tk graphics, and includes mechanisms for activating Tcl callbacks when signals change. As with many Tcl extensions, all *esim* commands are really sub-commands of a single Tcl command: *esim*. This

means that a sample *esim* command would be *esim load default.net*. This instructs the simulator to use the *esim* command interpreter, which then loads the netlist *default.net*.

3.1 Variable names and values

The simulator has access to all variables in the *esim* program that has been loaded. Because the *use* statement requires each instance to have a name unique within the component, each signal has a unique name composed of a period-separated list of component names followed by the signal name within the component. For example, consider the circuit shown in Figure 5. In the circuit, the top-level symbols are *topx* and *topy*. Each instance of the *bar* component creates its own copy of the symbols *x* and *y*. The *hi use* statement creates *hi.x* and *hi.y*, and the *lo use* statement creates *lo.x* and *lo.y*. Since each instance of *bar* itself creates two instances of *foo*, the following signals are defined: *lo.s.a*, *lo.s.b*, *lo.t.a*, *lo.t.b*, *hi.s.a*, *hi.s.b*, *hi.t.a*, and *hi.t.b*. Each of these signals may have its own value; even though there are 4 instances of the signal *a* in *foo*, each may have its own value (and thus has its own name) because each belongs to a different instance of the *foo* component.

Symbol values may be specified in either binary or hex (using the *-hex* or *-binary* switch to various *esim*

commands). The default for all commands is binary. Possible symbol values include 0, 1, X, and Z. 0 and 1 are high and low logic values. Z refers to a node that isn't currently driven by any gate. X means that the value of the node can't be determined. This can occur for several reasons. First, the node could be driven to different values (0 and 1) by two or more different gates at the same time. Second, the node could be driven by a single gate whose output value can't be determined for some reason, such as indeterminate inputs (X-valued). For binary, each bit is represented individually by one of these four symbols. For hex, however, nodes are shown in groups of four. If all nodes in a group of four have "real" values, a hex digit is displayed. If all are undriven, Z is displayed. Otherwise, X is displayed. This means that the four bits 010X would be shown as a single hex digit of X, as would 0Z11.

3.2 Controlling the simulation

The simulation is controlled by Tcl commands, each prefixed with the keyword *esim*. The simulator has commands to load a netlist, run for a set period of time, run for a fixed number of events, or simply run until the circuits have settled. The latter mode is not advised for complex systems with a clock; a line such as

```
clk <= ~clk after 100 ns
```

will never settle.

Running the simulation is not very useful without the ability to examine and set signals, and *esim* can do both. The *esim set* command can be used to force a signal, named as described in Section 3.1, to a particular value. This value is maintained even if the circuit tries to set it to a different value. The signal can be freed by issuing an *esim unset* command to allow the signal to once again vary normally. The *unset* command, however, does not erase the value to which the signal was set; it only *allows* it to change. For example, setting a clock to 0 and then issuing an *unset* command will initialize the clock to a fixed value and then allow it to vary over time.

The simulator also has commands for reading and writing memory. These commands are particularly useful for setting the contents of registers and cache

for simulations as well as for checking their contents after a simulation has run. The *esim read* and *esim write* command work in similar ways, and allow the writing of an arbitrary number of bits starting from a given bit offset. Unlike the *esim set* command, however, the *esim write* command is a one-time write of memory; future writes to the location by the circuit will succeed. The *esim write* command is particularly useful in scripts; a ten line script can read the contents of memory from a file and write them to a particular memory in the simulated circuit.

Perhaps the most useful feature of the simulation is the *esim trace* command. This command ties a Tcl variable's value to that of a signal in the circuit, updating the variable whenever the signal's value changes. For example, a command such as:

```
esim trace -hex a
```

will update the Tcl variable *Esim(a)* with the current value of signal *a* each time it changes. This feature is particularly useful when combined with the Tcl *trace* command; it allows Tcl to execute arbitrary code when a signal in the circuit changes. The trace on a signal can be removed with the *esim untrace* command.

The tracing feature is underutilized in the current *esim* distribution. There are simple routines to print the value of a signal when it changes, and there is a simple graphic display that can show the current values of various signals. However, we have not yet implemented more complex display mechanisms. In particular, *esim* cannot display signal values graphically, though this feature is planned for future releases.

4 Future work

In its current form, *esim* is a very powerful tool for teaching computer architecture. However, there are several additions and modifications that would significantly improve its usability. The first major change that we are currently working on is a rewrite of the core code for the application to use the standard template library (STL), removing the need for *libg++*. Though *libg++* was previously the best way to implement platform-independent code for common data structures, it has been supplanted by the

STL. The rewrite will improve performance by doing more to merge duplicate events in the event-driven simulator. This rewrite will also incorporate an interface generated by SWIG [3]. This change will standardize the Tcl/Tk interface code, improving the simulator's compatibility with Tcl and allowing it to work with other languages such as Python [4].

We are also planning to add graphical display of signal values and histories, taking advantage of the many graph widget packages available for Tk. This display will allow students to view the contents of signals and busses over time, and will resemble standard timing diagrams. By using the flexibility of Tcl, we can allow the simulator to display some or all of the signals, and can even use color and other cues to show "important" events in the display. Other possible features include zooming in and out to show shorter or longer time scales in the graph.

The *esim* language itself is stable, and we do not plan any major changes to it. One minor change we are considering is the addition of non-bitwise operators such as addition and subtraction. These operators make it easier to quickly design circuits; however, they do not correspond directly to simple digital logic components, making them less useful pedagogically.

5 Conclusions

We have designed and implemented a simple structural design language for use in undergraduate computer architecture classes. This language is easy for students to learn, and includes lacks the complex features necessary in VHDL and Verilog for building real hardware. By integrating the logic simulator into Tcl, *esim* allows students and instructors to use scripts and graphics to enhance the simulation and debugging process. As a result, *esim* should not be used to design actual hardware, but rather makes an excellent (and portable) pedagogical tool for teaching computer design to undergraduates.

The source code for the *esim* compiler and simulator is available from <http://www.csee.umbc.edu/~elm/esim/>. It has been compiled on Linux, and requires

flex, *bison*, *tcl*, and *libg++*. It is currently being rewritten to use Standard Template Library classes in place of *libg++*.

References

- [1] Peter J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann, 1996.
- [2] Peter J. Ashenden, *The Student's Guide to VHDL*, Morgan Kaufmann, 1998.
- [3] David M. Beazley, "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++," *4th Annual Tcl/Tk Workshop*, Monterey, CA, July 1996, pages 129-139.
- [4] John E. Grayson, *Python and Tkinter Programming*, Manning Publications, 2000.
- [5] Samir Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, Prentice Hall, 1996.
- [6] Brent B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, 1999.