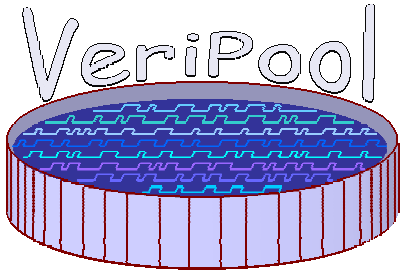


Verilator & Internals

July 2005
Philips Semiconductor

Wilson Snyder,

SiCortex, Inc.
wsnyder@wsnyder.org

http://www.veripool.com

Agenda

- Preliminary Diversion – Verilog-Mode
- Introduction & History
- Using Verilator
  - Example Translations
  - SystemPerl
  - Verilog Coding Tips
- Verilator Internals and Debugging
  - Data structures
  - Debugging tips
- Futures
  - Future Features
  - You can help
- Conclusion & Getting the Tools

Verilog-Mode for Emacs

- Ralf asked for a summary of my other tools, and this is my most popular,
- So, before we talk about a compiler, let's talk about making Verilog coding more productive.
- (This is an excerpt of a larger presentation on www.veripool.com.)

Verilog-Mode for Emacs

- Verilog requires lots of typing to hook up signals, ports, sensitivity lists, etc.
- Verilog-Mode saves you from typing them
  - Fewer lines of code means less development, fewer bugs.
- Not a preprocessor, but all input & output code is always completely "valid" Verilog.
  - Can always edit code without the program.
- Batch executable for non-Emacs users.

Verilog-mode Meta-Comments

With this key sequence,
Verilog-Mode parses the verilog code, and
expands the text after any /*AUTO*/ comments.

```
module (/*AUTOARG*/)

input   a;
input   ena;
output  z;

always @(/*AS*/)
  z = a & ena;
```

GNU Emacs (Verilog-Mode)

C-c C-a
(or use menu)

C-c C-d
(or use menu)

```
module (/*AUTOARG*/
        // Outputs
        z,
        // Inputs
        a, ena);

input   a;
input   ena;
output  z;

always @(/*AS*/a or ena)
  z = a & ena;
```

GNU Emacs (Verilog-Mode)

Automatic Wires

/*AUTOWIRE*/ takes the outputs of sub modules and declares wires for them (if needed -- you can declare them yourself).

```
 …
/*AUTOWIRE*/
/*AUTOREG*/

a a (// Outputs
     .bus   (bus[0]),
     .z     (z));

b b (// Outputs
     .bus   (bus[1]),
     .y     (y));
```

GNU Emacs (Verilog-Mode)

```
/*AUTOWIRE*/
// Beginning of autos
wire [1:0] bus; // From a,b
wire       y;   // From b
wire       z;   // From a
// End of automatics

/*AUTOREG*/

a a (
     // Outputs
     .bus   (bus[0]),
     .z     (z));

b b (
     // Outputs
     .bus   (bus[1]),
     .y     (y));
```

GNU Emacs (Verilog-Mode)

Automatic Registers


```
…
output [1:0] from_a_reg;
output       not_a_reg;

/*AUTOWIRE*/
/*AUTOREG*/

wire not_a_reg = 1'b1;
```

GNU Emacs (Verilog-Mode)

/*AUTOREG*/ saves having to duplicate reg statements for nets declared as outputs. (If it's declared as a wire, it will be ignored, of course.)

```
output [1:0] from_a_reg;
output       not_a_reg;

 /*AUTOWIRE*/
 /*AUTOREG*/
 // Beginning of autos
 reg [1:0] from_a_reg;
 // End of automatics

 wire not_a_reg = 1'b1;

 always

   … from_a_reg = 2'b00;
```

GNU Emacs (Verilog-Mode)

Simple Instantiations

/*AUTOINST*/
Look for the submod.v file, read its in/outputs.

```
submod s (/*AUTOINST*/);
```

```
module submod;
  output out;
  input in;

  …

endmodule
```

GNU Emacs (Verilog-Mode)

```
submod s (/*AUTOINST*/
          // Outputs
          .out  (out),
          // Inputs
          .in   (in));
```

Keep signal names consistent!

Note the simplest and most obvious case is to have the signal name on the upper level of hierarchy match the name on the lower level. Try to do this when possible.

Occasionally two designers will interconnect designs with different names. Rather then just connecting them up, it's a 30 second job to use *vrename* from my website to make them consistent.

Exceptions to Instantiations

Method 1: AUTO_TEMPLATE lists exceptions for "submod." The ports need not exist.
(This is better if submod occurs many times.)

```
/* submod AUTO_TEMPLATE (
   .z (otherz),
   );
*/

submod s (
          .a (except1),
          /*AUTOINST*/);
```

GNU Emacs (Verilog-Mode)

```
/* submod AUTO_TEMPLATE (
   .z (otherz),
   );
*/

submod s (
          .a  (except1),
          /*AUTOINST*/
          .z  (otherz),
          .b  (b));
```

GNU Emacs (Verilog-Mode)

Method 2: List the signal before the AUTOINST.

Signals not mentioned otherwise are direct connects.

Multiple Instantiations

@ in the template takes the trailing digits from the reference.

```
/* submod AUTO_TEMPLATE (
   .z (out[@]),
   .a (invec@[]));
*/

submod i0 (/*AUTOINST*/);

submod i1 (/*AUTOINST*/);

submod i2 (/*AUTOINST*/);
```

GNU Emacs (Verilog-Mode)

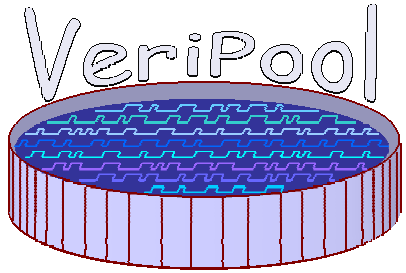
[] takes the bit range for the bus from the referenced module. Other methods allow regexps, and lisp program templates.

```
/* submod AUTO_TEMPLATE (
   .z (out[@]),
   .a (invec@[]));
*/

submod i0 (/*AUTOINST*/
           .z (out[0]),
           .a (invec0[31:0]));

submod i1 (/*AUTOINST*/
           .z (out[1]),
           .a (invec1[31:0]));
```

GNU Emacs (Verilog-Mode)

More Autos & Verilog-Mode Conclusion

- Other Automatics
  - AUTOASCIIENUM – Create ascii decodings for state machine states.
  - AUTOINOUTMODULE – Pull all I/O from another module for shells
  - AUTOOUTPUT – Make outputs for all signals
  - AUTORESET – Reset all signals in a flopped block
  - Goto-module – Goto any module name in your design with 2 keys.
- Verilog-mode allows for faster coding with less bugs.
- There's NO disadvantages to using it.
  - Many IP Vendors, including MIPS and ARM use Verilog-Mode.
- For More Information
  - See presentations, papers and the distribution on http://www.veripool.com



Verilator
Introduction & History

Introduction

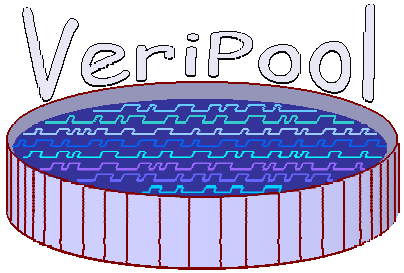
- Verilator was born to connect Verilog to C/C++.
  - Verilog is the Synthesis Language.
  - C++ is generally the embedded programming language.
  - And C++ is often the Test-bench Language.
  - Of the 15 chip projects I've worked on, all but two had C++ test benches with Verilog cores.
- So, throw away the Verilog testing constructs, and let's synthesize the Verilog into C++ code.
- Your simulator model can now be 100% C++!
  - Enables easy cross compiling, gdb, valgrind, lots of other tools.

History (1 of 2)

- 1994 – Digital Semiconductor
  - We were deciding on the methodology for a new Core Logic chipset. We wanted Verilog for Synopsys synthesis. We already had a C-based simulation system for our CPU, so Paul Wasson decided he would write a program to directly translate Synthesizable Verilog into C.
    - Verilator 1.0 maps Verilog almost statement-to-statement to C. We let the C compiler do all optimizations.
- 1997/8/9 – Intel Network Processors
  - Duane Galbi starts using Verilator at Intel, and takes over the source.
    - Verilator simplifies Boolean math expressions (2x faster).
    - Verilator released into the public domain.
- 2001 – Maker (Conexant/Mindspeed)
  - After a few years, I take the sources from Duane.

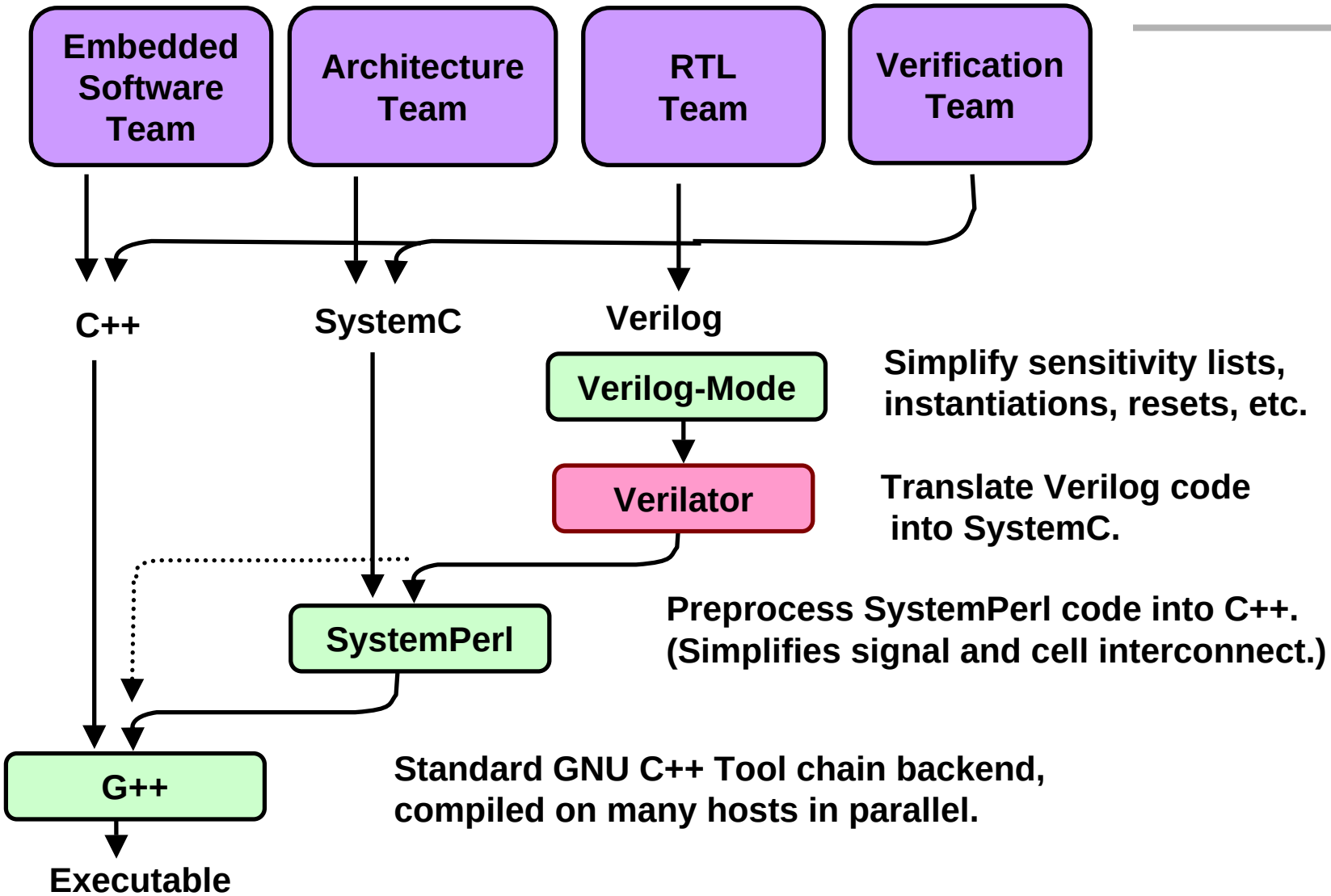
History (2 of 2)

- 2001/2 – Nauticus Networks
  - At my new startup, we decide to try a new modeling language, SystemC. Verilator gets adopted to enable easy substitution of Verilog into SystemC.
    - Verilator 2.1.0 outputs first SystemC code.
- 2003/4 – Nauticus/Sun Microsystems
  - Verilator gets a complete rewrite and moves from translation to a full compiler.
    - Verilator 3.000 adds std compiler optimizations. (3x faster)
    - Verilator 3.012 has first patch from a outside contributor.
    - Verilator 3.201 orders all code. (2x faster & beats VCS.)
- 2004/5 – SiCortex
  - I join my third startup, and we settle on a SystemC methodology. SiCortex starts purchasing IP, so Verilator grows to support most Verilog 2001 features.
    - Verilator tops 100 downloads per month (excluding robots, etc)



Using Verilator

Typical Verilator CAD Flow



Verilator is a Compiler

- Verilator compiles Synthesizable Verilog into C++
  - Always statements, wires, etc.
  - No time delays (a <= #{n} b;)
  - Only two state simulation (no tri-state busses).
  - Unknowns are randomized (even better than having Xs).
  - All clocks from primary inputs (well, some generated might be ok…)
- Creates a "pure" C++/SystemC wrapper around the design
- Creates own internal interconnect and signal formats
  - Version 2.0.0 tried sc_signals, but they are >>10x slower!
  - Plays several tricks to get good, fast code out of gcc.

Example Translated to C++

- The top wrapper looks similar to the top Verilog module.
- Inputs and outputs map directly to bool, uint32_t, uint64_t, or array of uint32_t's:

```
module Convert;
   input clk
   input [31:0] data;
   output [31:0] out;

   always @ (posedge clk)
      out <= data;
endmodule
```



```
#include "verilated.h"

class Convert {
   bool      clk;
   uint32_t  data;
   uint32_t  out;

   void eval();
}
```

Calling the model

- You generally call the Verilated class inside a simple loop.

```
int main() {
    Convert* top = new Convert();
    while (!Verilated::gotFinish()) {
        top->data = …;
        top->clk = !top->clk;

        top->eval();

        … = top->out();

        // Advance time…
    }
}
```

```
class Convert {
   bool      clk;
   uint32_t  data;
   uint32_t  out;

   void eval();
}
```

SystemPerl

- Verilator can output a dialect of SystemC, SystemPerl.
  - This matches SystemPerl code hand-written by our architects and verifiers.
  - SystemPerl translates the files into standard SystemC code we compile.
  - Similar to Verilog-Mode for Emacs.

- SystemPerl makes SystemC faster to write and execute
  - My last project wrote only 43% as many SystemC lines due to SysPerl.
  - SystemPerl standardizes Pin and Cell interconnect,
  - Lints interconnect (otherwise would get ugly run-time error),
  - Automatically connects sub-modules in “shell” modules, ala AUTOINST,
  - And adds “#use” statements for finding and linking library files.

- Reducing code means faster development
  - And less debugging!

Example Translated to SystemPerl

- Inputs and outputs map directly to bool, uint32_t, uint64_t, or sc_bv's.
- Similar for pure SystemC output.

```
module Convert;
   input clk
   input [31:0] data;
   output [31:0] out;

   always @ (posedge clk)
      out <= data;
endmodule
```



```
#include "systemperl.h"
#include "verilated.h"

SC_MODULE(Convert) {
   sc_in_clk         clk;
   sc_in<uint32_t>   data;
   sc_out<uint32_t> out;

   void eval();
}

SP_CTOR_IMP(Convert) {
  SP_CELL(v,VConvert);
  SC_METHOD(eval);
  sensitive(clk);
}
…
```

Talking C++ inside Verilog (1 of 2: Public Functions)

- Verilator allows tasks/functions to be called from “above” C++
  - We typically use this to have “0 time” configuration of the chip registers.
  - Some performance impact, but far less then what a commercial tool does.

```
module Pub;
   …
   reg state;

   task setState;
        input flag;
        // verilator public
        state = flag;
   endtask

   function getState;
        // verilator public
        getState = state;
   endfunction
```

Declare the function/task as visible to the C++ world.

```
SC_MODULE(Pub) {
   …
   void setState(bool flag);
   bool getState();
}
```

Talking C++ inside Verilog (2 of 2: Embedding)

- Verilator allows C++ code to be embedded directly in Verilog

```
`systemc_include
  #include "MDebug.h"

`systemc_header
  public:
       int debug();

`systemc_ctor
  __message = MDebug::debug();

`systemc_implementation
  int debug() {
     return __message;
  }
`verilog
  always @ (posedge clk)
     if ($c1("debug()"))
         $write("Debug message…\n");
```

Place at the top of the generated header file.

Place inside the class definition of the generated header file.

Place in the constructor of the generated C++ file.

Place in the generated C++ file.

Use the C++ text "debug()" that returns a one bit value for this expression.

Verilog Coding Tips

- Lint your code
  - Verilator doesn't check for many common coding mistakes.
    - For example, assigning a input found a bug which dumped core.
- Don't ignore warnings
  - Verilator gets limited testing of cases where they are suppressed.
  - Generally, it doesn't complain about reasonable constructs
    - wire [2:0] width_mismatch = 0; // This is fine!
  - Ignoring "unoptimizable" warnings can drop performance by 2x.
- Split up always statements for performance
  - Put as few statements as reasonable in a combo or seq always block.
  - This allows it to better order the code.
- Gate clocks in common module and ifdef
  - Any ugly constructs should be put into a cell library, and ifdef'ed into a faster Verilator construct.

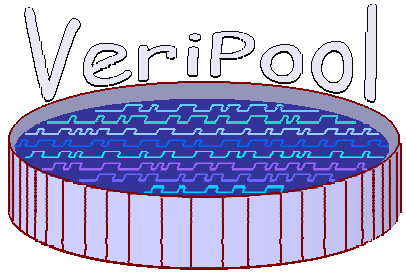
Verilator and Commercial Tools

- Verilator wasn't intended as a substitute for a commercial simulator, but the best way to get the job done. Until recently, all the commercial tools required slow PLI code, and any mixing of C++ and Verilog was very painful.

- I've never used Verilator as a sign-off simulator.
  - Last project, we captured the pins from a Verilator run, and replayed against a gate level model running on a commercial simulator.

- Though we certainly buy fewer simulator licenses.
  - 80% of our simulation cycles are under Verilator.
  - It's not so much to save money - the Verilated model builds 1.7 times faster, and runs 2.1 times faster.

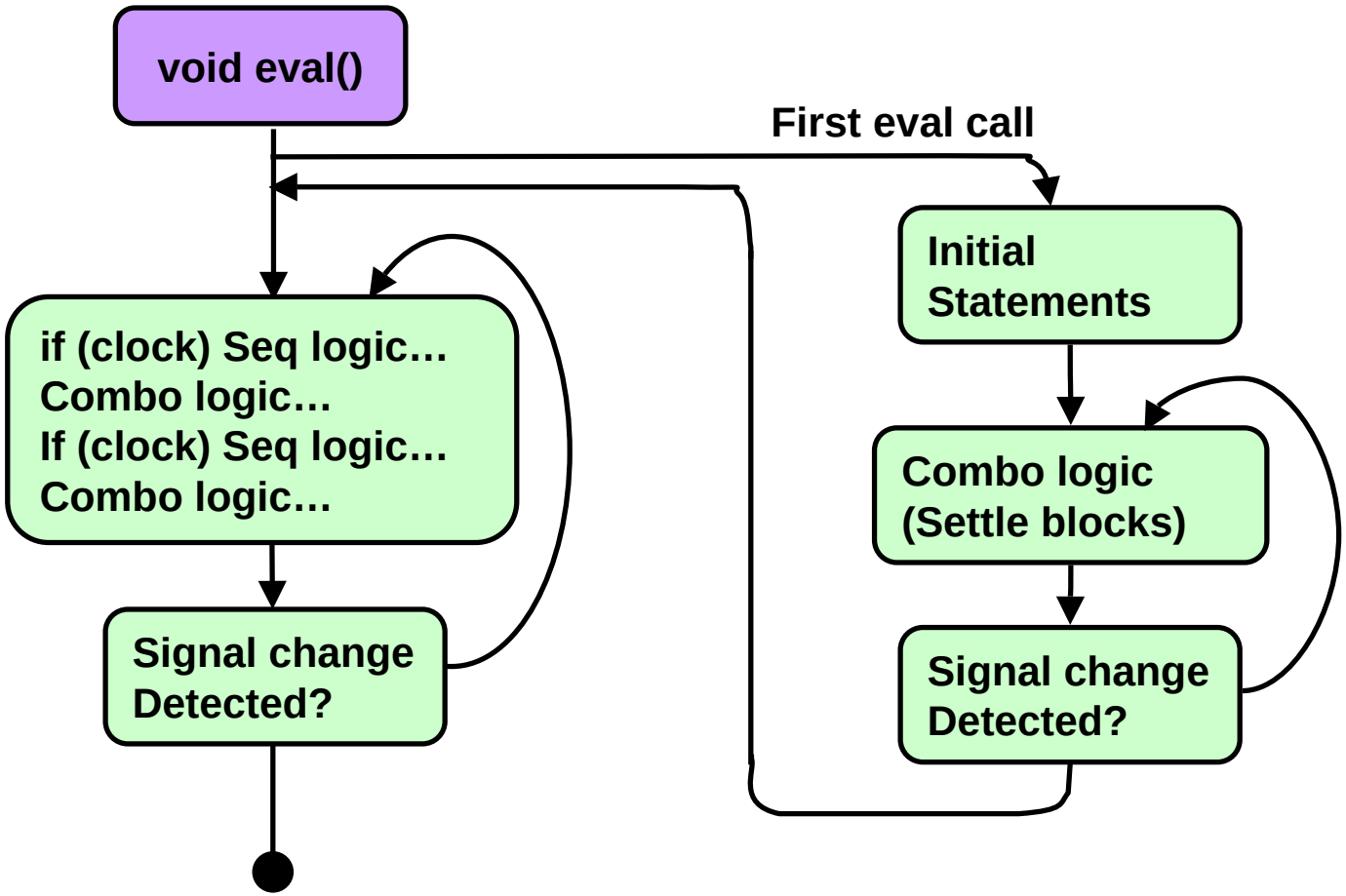
- Fortunately the vendors ignore me.
  - Until the first big EE Times article? ☺



Verilator Internals and Debugging

Internal Loop

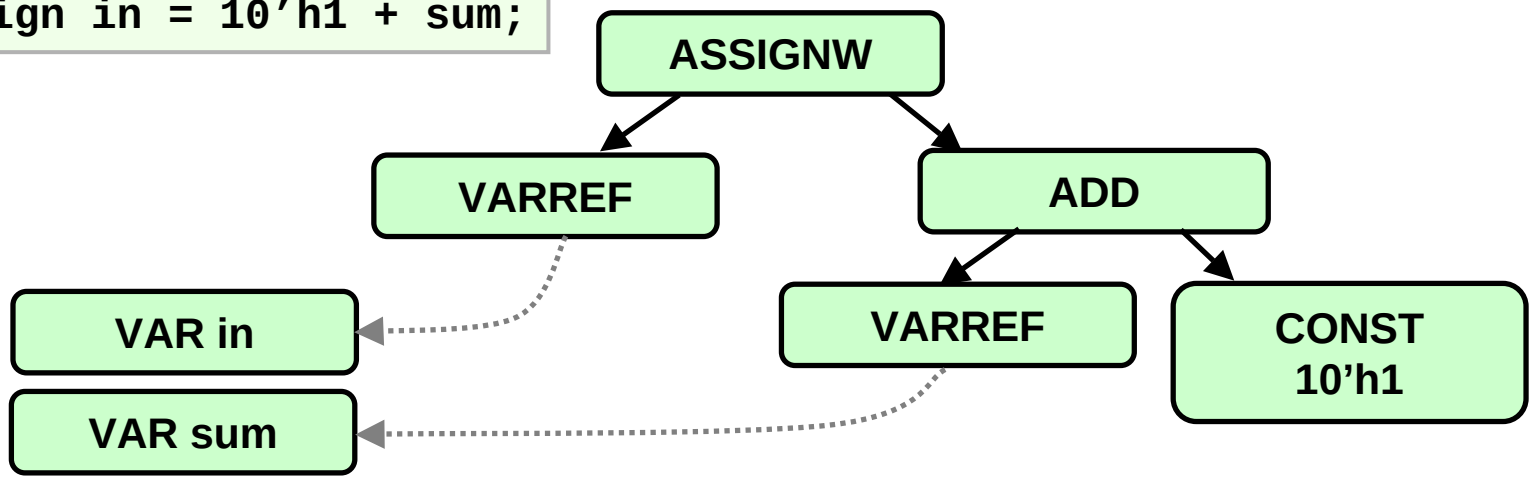
- Inside the model are two major loops, the settle loop called at initialization time, and the main change loop.



AstNode

- The core data structure is a AstNode, forming a tree

```
assign in = 10'h1 + sum;
```



- If you run with –debug, you'll see this in a .tree file:

```
1:1: ASSIGNW 0x8add8 {37} w10
1:1:1: ADD 0x8ad70 {37} w10
1:1:1:1: VARREF 0x8abd8 {37} w10  in [RV] <- VAR in
1:1:1:2: CONST 0x8acc0 {37} w10  10'h1
1:1:2: VARREF 0x8ab30 {37} w10  sum [LV] => VAR sum
```

Source Code Line Number.

Node Address

LV indicates a lvalue – it sets the variable.

Visitor Template

- Most of the code is written as transforms on the tree
- Uses the C++ “Visitor Template”

```
class TransformVisitor : AstNVisitor {

   virtual void visit(AstAdd* nodep) {
      // hit an add
      nodep->iterateChildren(*this);
   }
   virtual void visit(AstNodeMath* nodep)
      // A math node (but not called on an
      // AstAdd because there is a
      // visitor for AstAdd above.
      nodep->iterateChildren(*this);
   }
   virtual void visit(AstNode* nodep) {
      // Default catch-all
      nodep->iterateChildren(*this);
   }
}
```

The node-type overloaded visit function is called each time a node of type AstAdd is seen in the tree.

Recurse on all nodes below this node; calling the visit function based on the node type.

AstNodeMath is the base class of all math nodes. This will get called if a math node visitor isn’t otherwise defined.

Compile Phases (1 of 4: Module based)

- Link.
  - Resolve module and signal references.
- Parameterize
  - Propagate parameters, and make a unique module for each set of parameters.
- Expression Widths
  - Determine expression widths. (Surprisingly complicated!)
- Coverage Insertion (Optional)
  - Insert line coverage statements at all if, else, and case conditions.
- Constification & Dead code elimination
  - Eliminate constants and simplify expressions
  - (“if (a) b=1; else b=c;” becomes “b = a|c”;)

Compile Phases (2 of 4: Module based)

- Task Inlining, Loop Unrolling, Module Inlining (Optional)
  - Standard compiler techniques.
- Tristate Elimination
  - Convert case statements into if statements
  - Replace assignments to X with random values. (Optional)
- Always Splitting (Optional)
  - Break always blocks into pieces to aid ordering.
    - always @ ... begin a<=z; b<=a; end

Compile Phases (3 of 4: Scope Based)

- Scope
  - If a module is instantiated 5 times, make 5 copies of its variables.
- Gate (Optional)
  - Treat the design as a netlist, and optimize away buffers, inverters, etc.
- Delayed
  - Create delayed assignments.
    - a<=a+1 becomes “a_dly=a; ….; a_dly=a+1; …. a=a_dly;”
- Ordering
  - Determine the execution order of each always/wire statement.
    - This may result in breaking combinatorial feedback paths.
    - The infamous “not optimizable” warning
- Lifetime Analysis (Optional)
  - If a variable is never used after being set, no need to set it.
  - If code ordering worked right, this eliminates _dly variables.

Compile Phases (4 of 4: Backend)

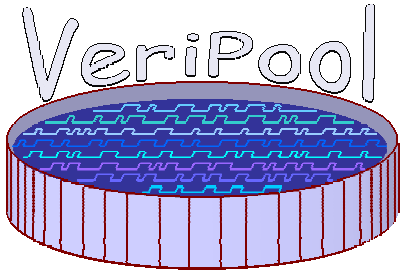
- Descope
  - Reverse the flattening process; work on C++ classes now.
- Clean
  - Add AND statements to correct expression widths.
  - If a 3 bit value, then a + 1 in C++ needs to become (a+1) & 32'b111;
- Expand & Substitute (Optional)
  - Expand internal operations (bit selects, wide ops) into C++ operations.
  - Substitute temporaries with the value of the temporary.
- Output code

Graph Optimizations

- Some transformations rely heavily on graphs.
- Always Block Splitting
  - Vertexes are each statement in the always block
  - Edges are the variables in common between the statements
  - Any weakly connected sub-graphs indicate independent statements.
- Gate Optimization
  - Just like a netlist, Vertexes are wires and logic
  - Edges connect each wire to the logic it receives/drives
  - We can then eliminate eliminate buffers, etc.
- Ordering
  - Vertexes are statements and variables
  - Directional edges connect the statements to relevant variables
  - Break edges if there are any loops
  - Enumerate the vertexes so there are no backward edges

Debugging Verilated code

- Sorry.
- Run with –debug
  - This enables internal checks, which may fire a internal assertion.
  - It also dumps the internal trees.
- Try -O[each lower case letter]
  - This will disable different optimization steps, and hopefully isolate it.
- Make a standalone test_regress example
  - This will allow me to add it to the suite. See the verilator manpage.
- (Experts) Look at the generated source
  - Examine the output .cpp code for the bad section.
  - Then, walk backwards through the .tree files, searching for the node pointer, and find where it got mis-transformed.
  - That generally isolates it down to a few dozen lines of code.



Futures

Future Language Enhancements

Generally, new features are added as requested, unless they are difficult ☺

- Support Generated Clocks (correctness, not speed)
  - Most of the code is there, but stalled release as too hard to debug.

★ Assertions (PSL or SVL)
  - Especially need testcases, and an assertion generator.
  - If anyone has experience translating or optimizing assertions, let me know.

- Signed Numbers
  - Often requested, but lots of work and not valuable to my employer.
  - Now even “integers” are unsigned. Surprisingly, this rarely matters.

Future Performance Enhancements (1 of 2)

- Eliminate duplicate logic (commonly added for timing.)
  - "wire a=b|c; wire a2=b|c;"
- Gated clock optimizations
  - Philips is transforming some input code into mux-flops.
- Latch optimizations
  - Not plentiful, but painful when they occur.
- Cache optimizations
  - Verilator optimizes some modules so well, it becomes load/store limited.
  - Need ideas for eliminating load/stores and cache packing.
- Reset optimization
  - Lots of "if reset…" code that doesn't matter after a few dozen cycles.

Future Performance Enhancements (2 of 2: Dreams)

- Embed SystemPerl and schedule SystemC code.
  - Profiling shows some large tests spend more time in SystemC interconnect routines then in the Verilated code.
- Multithreaded execution
  - Multithreaded/multicore CPUs are now commodities.

Tool Development

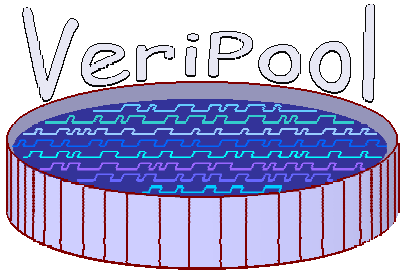
- FAQ: How much time do you spend on this?
  - Verilator currently averages about a day a week.
  - All my other tools add up to at most another day or so – though most of these changes are to directly support our verification and design team.
  - Rest of time I'm doing architecture and RTL for my subchip.
- I have help
  - Major testing by Ralf Karge here at Philips, and Jeff Dutton at Sun.
- FAQ: How can I get "away" with spending time on this?
  - When choosing a job, I've obtained a contractual agreement my tools may become public domain.
    - Sort of a "reverse NDA" – Imagine if this were the general practice.
  - In trade, I bring a productivity increase to the organization.
  - Public feedback makes the tools better.
    - A user contributed 64 bit patches, two months later I'm on a 64 bit OS.

You Can Help (1 of 2: Testing, testing, testing…)

- Firstly, I need more testcases!
  - Many enhancements are gated by testing and debugging.
- Large standalone test cases
  - Need a large testchips and **simple** testbenches.
  - Add a tracing and cycle-by-cycle shadow simulation mode, so finding introduced bugs is greatly simplified?
  - Port and run all opencores.org code?
- Need better random Verilog program generator
  - Now, most arithmetic bugs are found with vgen, a random math expression generator.
  - It commonly finds bugs in other simulators and GCC.
  - This needs to be extended/rewritten to test the rest of the language.
  - A great use for a idle verification engineer – know one?
- Improve the graph viewer or find another (Java)

You Can Help (2 of 2)

- Run gprof/oprofile and tell me your bottlenecks
  - Most optimizations came from “oh, this looks bad”
- Tell me what changes you’d like to see
  - I don’t hear from most users, and have no idea what they find frustrating.
- Advocate.
- Of course, patches and co-authors always wanted!
  - The features listed before, or anything else you want.



Conclusion

Conclusions

- Verilator promotes Faster RTL Development
  - Verilog is standard language, and what the downstream tools want.
  - Waveforms “look” the same across RTL or SystemC, no new tools.

- And faster Verification Development
  - Trivial integration with C++ modeling.
  - Every component can each be either behavioral or RTL code.
  - C++ hooks can be added to the Verilog
  - Automatic coverage analysis

- License-Free runtime is good
  - Easy to run on laptops or SW developer machines.
  - Run as fast as major simulators.
  - $$ we would spend on simulator runtime licenses may go to computes.

Available from Veripool.com

- Verilator:
  - GNU Licensed
  - C++ and Perl Based, Windows and Linux
  - http://www.veripool.com

- Also free on my site:
  - Dinotrace – Waveform viewer w/Emacs annotation.
  - Make::Cache - Object caching for faster compiles.
  - Schedule::Load – Load balancing (ala LSF).
  - Verilog-Mode - /*AUTO…*/ expansion, highlighting.
  - Verilog-Perl – Verilog Perl preprocessor and signal renaming.
  - Voneline – Reformat gate netlists for easy grep and editing.
  - Vpm – Add simple assertions to any Verilog simulator.
  - Vregs – Extract register and class declarations from documentation.
  - Vrename – Rename signals across many files (incl SystemC files).