Verilator 4.006 Internals Manual

http://www.veripool.org

2018-10-27

Contents

1 NAME

Verilator Internals

2 INTRODUCTION

This file discusses internal and programming details for Verilator. It's the first for reference for developers and debugging problems.

See also the Verilator internals presentation at http://www.veripool.org.

3 CODE FLOWS

Verilator Flow

The main flow of Verilator can be followed by reading the Verilator.cpp process() function:

First, the files specified on the command line are read. Reading involves preprocessing, then lexical analysis with Flex and parsing with Bison. This produces an abstract syntax tree (AST) representation of the design, which is what is visible in the .tree files described below.

Verilator then makes a series of passes over the AST, progressively refining and optimizing it.

Cells in the AST first linked, which will read and parse additional files as above.

Functions, variable and other references are linked to their definitions.

Parameters are resolved and the design is elaborated.

Verilator then performs many additional edits and optimizations on the hierarchical design. This includes coverage, assertions, X elimination, inlining, constant propagation, and dead code elimination.

References in the design are then pseudo-flattened. Each module's variables and functions get "Scope" references. A scope reference is an occurrence of that unflattened variable in the flattened hierarchy. A module that occurs only once in the hierarchy will have a single scope and single VarScope for each variable. A module that occurs twice will have a scope for each occurrence, and two VarScopes for each variable. This allows optimizations to proceed across the flattened design, while still preserving the hierarchy.

Additional edits and optimizations proceed on the pseudo-flat design. These include module references, function inlining, loop unrolling, variable lifetime analysis, lookup table creation, always splitting, and logic gate simplifications (pushing inverters, etc).

Verilator orders the code. Best case, this results in a single "eval" function which has all always statements flowing from top to bottom with no loops.

Verilator mostly removes the flattening, so that code may be shared between multiple invocations of the same module. It localizes variables, combines identical functions, expands macros to C primitives, adds branch prediction hints, and performs additional constant propagation.

Verilator finally writes the C++ modules.

Key Classes Used in the Verilator Flow

`AstNode`

The AST is represented at the top level by the class `AstNode`. This abstract class has derived classes for the individual components (e.g. `AstGenerate` for a generate block) or groups of components (e.g. `AstNodeFTask` for functions and tasks, which in turn has `AstFunc` and `AstTask` as derived classes).

Each `AstNode` has pointers to up to four children, accessed by the `op1p` through `op4p` methods. These methods are then abstracted in a specific Ast* node class to a more specific name. For example with the `AstIf` node (for `if` statements), `ifsp` calls `op2p` to give the pointer to the AST for the "then" block, while `elsesp` calls `op3p` to give the pointer to the AST for the "else" block, or NULL if there is not one.

`AstNode` has the concept of a next and previous AST - for example the next and previous statements in a block. Pointers to the AST for these statements (if they exist) can be obtained using the `back` and `next` methods.

It is useful to remember that the derived class `AstNetlist` is at the top of the tree, so checking for this class is the standard way to see if you are at the top of the tree.

By convention, each function/method uses the variable `nodep` as a pointer to the `AstNode` currently being processed.

`AstNVisitor`

The passes are implemented by AST visitor classes (see Visitor Functions). These are implemented by subclasses of the abstract class, `AstNVisitor`. Each pass creates an instance of the visitor class, which in turn implements a method to perform the pass.

`V3Graph`

A number of passes use graph algorithms, and the class `V3Graph` is provided to represent those graphs. Graphs are directed, and algorithms are provided to manipulate the graphs and to output them in *GraphViz* dot format (see http://www.graphviz.org/). `V3Graph.h` provides documentation of this class.

`V3GraphVertex`

This is the base class for vertices in a graph. Vertices have an associated `fanout`, `color` and `rank`, which may be used in algorithms for ordering the graph. A generic `user`/`userp` member variable is also provided.

Virtual methods are provided to specify the name, color, shape and style to be used in dot output. Typically users provide derived classes from `V3GraphVertex` which will reimplement these methods.

Iterators are provided to access in and out edges. Typically these are used in the form:

```
for (V3GraphEdge *edgep = vertexp->inBeginp();
     edgep;
     edgep = edgep->inNextp()) {
```

`V3GraphEdge`

This is the base class for directed edges between pairs of vertices. Edges have an associated `weight` and may also be made `cutable`. A generic `user`/`userp` member variable is also provided.

Accessors, `fromp` and `top` return the "from" and "to" vertices respectively.

Virtual methods are provided to specify the label, color and style to be used in dot output. Typically users provided derived classes from `V3GraphEdge` which will reimplement these methods.

`V3GraphAlg`

This is the base class for graph algorithms. It implements a `bool` method, `followEdge` which algorithms can use to decide whether an edge is followed. This method returns true if the graph edge has weight greater than one and a user function, `edgeFuncp` (supplied in the constructor) returns `true`.

A number of predefined derived algorithm classes and access methods are provided and documented in `V3GraphAlg.cpp`.

Multithreaded Mode

In --threads mode, the frontend of the Verilator pipeline is the same as serial mode, up until V3Order.

V3Order builds a fine-grained, statement-level dependency graph that governs the ordering of code within a single eval() call. In serial mode, that dependency graph is used to order all statements into a total serial order. In parallel mode, the same dependency graph is the starting point for a partitioner (V3Partition).

The partitioner's goal is to coarsen the fine-grained DAG into a coarser DAG, while maintaining as much available parallelism as possible. Often the partitioner can transform an input graph with millions of nodes into a coarsened execution graph with a few dozen nodes, while maintaining enough parallelism to take advantage of a modern multicore CPU. Runtime synchronization cost is not prohibitive with so few nodes.

Partitioning

Our partitioner is similar to the one Vivek Sarkar described in his 1989 paper "Partitioning and Scheduling Parallel Programs for Multiprocessors".

Let's define some terms:

`Par Factor`

: The available parallelism or "par-factor" of a DAG is the total cost to execute all nodes, divided by the cost to execute the longest critical path through the graph. This is the speedup you would get from running the graph in parallel, if given infinite CPU cores available and communication and synchronization are zero.

`Macro Task`

: When the partitioner coarsens the graph, it combines nodes together. Each fine-grained node represents an atomic "task"; combined nodes in the coarsened graph are "macro-tasks". This term comes from Sarkar. Each macro-task executes from start to end on one processor, without any synchronization to any other macro-task during its execution. (Synchronization only happens before the macro-task begins or after it ends.)

`Edge Contraction`

: Our partitioner, like Sarkar's, primarily relies on "edge contraction" to coarsen the graph. It starts with one macro-task per atomic task and iteratively combines pairs of edge-connected macro-tasks.

`Local Critical Path`

: Each node in the graph has a "local" critical path. That's the critical path from the start of the graph to the start of the node, plus the node's cost, plus the critical path from the end of the node to the end of the graph.

Sarkar calls out an important trade-off: coarsening the graph reduces runtime synchronization overhead among the macro-tasks, but it tends to increase the critical path through the graph and thus reduces par-factor.

Sarkar's partitioner, and ours, chooses pairs of macro-tasks to merge such that the growth in critical path is minimized. Each candidate merge would result in a new node, which would have some local critical path. We choose the candidate that would produce the shortest local critical path. Repeat until par-factor falls to a target threshold. It's a greedy algorithm, and it's not guaranteed to produce the best partition (which Sarkar proves is NP-hard).

Estimating Logic Costs

To compute the cost of any given path through the graph, Verilator estimates an execution cost for each task. Each macro-task has an execution cost which is simply

the sum of its tasks' costs. We assume that communication overhead and synchronization overhead are zero, so the cost of any given path through the graph is simply the sum of macro-task execution costs. Sarkar does almost the same thing, except that he has nonzero estimates for synchronization costs.

Verilator's cost estimates are assigned by the InstrCountCostVisitor. This class is perhaps the most fragile piece of the multithread implementation. It's easy to have a bug where you count something cheap (eg. accessing one element of a huge array) as if it were expensive (eg. by counting it as if it were an access to the entire array.) Even without such gross bugs, the estimates this produce are only loosely predictive of actual runtime cost. Multithread performance would be better with better runtime costs estimates. This is an area to improve.

Scheduling Macro-Tasks at Runtime

After coarsening the graph, we must schedule the macro-tasks for runtime. Sarkar describes two options: you can dynamically schedule tasks at runtime, with a runtime graph follower. Sarkar calls this the "macro-dataflow model." Verilator does not support this; early experiments with this approach had poor performance.

The other option is to statically assign macro-tasks to threads, with each thread running its macro-tasks in a static order. Sarkar describes this in Chapter 5. Verilator takes this static approach. The only dynamic aspect is that each macro task may block before starting, to wait until its prerequisites on other threads have finished.

The synchronization cost is cheap if the prereqs are done. If they're not, fragmentation (idle CPU cores waiting) is possible. This is the major source of overhead in this approach. The --prof-threads switch and the `verilator_gantt` script can visualize the time lost to such fragmentation.

Locating Variables for Best Spatial Locality

After scheduling all code, we attempt to locate variables in memory such that variables accessed by a single macro-task are close together in memory. This provides "spatial locality" -- when we pull in a 64-byte cache line to access a 2-byte variable, we want the other 62 bytes to be ones we'll also likely access soon, for best cache performance.

This turns out to be critical for performance. It should allow Verilator to scale to very large models. We don't rely on our working set fitting in any CPU cache; instead we essentially "stream" data into caches from memory. It's not literally streaming, where the address increases monotonically, but it should have similar performance characteristics, so long as each macro-task's dataset fits in one core's local caches.

To achieve spatial locality, we tag each variable with the set of macro-tasks that access it. Let's call this set the "footprint" of that variable. The variables in a given module have a set of footprints. We can order those footprints to minimize the distance

between them (distance is the number of macro-tasks that are different across any two footprints) and then emit all variables into the struct in ordered-footprint order.

The footprint ordering is literally the traveling salesman problem, and we use a TSP-approximation algorithm to get close to an optimal sort.

This is an old idea. Simulators designed at DEC in the early 1990s used similar techniques to optimize both single-thread and multi-thread modes. (Verilator does not optimize variable placement for spatial locality in serial mode; that is a possible area for improvement.)

Improving Multithreaded Performance Further (a TODO list)

`Wave Scheduling`

To allow the verilated model to run in parallel with the testbench, it might be nice to support "wave" scheduling, in which work on a cycle begins before eval() is called or continues after eval() returns. For now all work on a cycle happens during the eval() call, leaving Verilator's threads idle while the testbench (everything outside eval()) is working. This would involve fundamental changes within the partitioner, however, it's probably the best bet for hiding testbench latency.

`Efficient Dynamic Scheduling`

To scale to more than a few threads, we may revisit a fully dynamic scheduler. For large (>16 core) systems it might make sense to dedicate an entire core to scheduling, so that scheduler data structures would fit in its L1 cache and thus the cost of traversing priority-ordered ready lists would not be prohibitive.

`Static Scheduling with Runtime Repack`

We could modify the static scheduling approach by gathering actual macro-task execution times at run time, and dynamically re-packing the macro-tasks into the threads also at run time. Say, re-pack once every 10,000 cycles or something. This has the potential to do better than our static estimates about macro-task run times. It could potentially react to CPU cores that aren't performing equally, due to NUMA or thermal throttling or nonuniform competing memory traffic or whatever.

`Clock Domain Balancing`

Right now Verilator makes no attempt to balance clock domains across macro-tasks. For a multi-domain model, that could lead to bad gantt chart fragmentation. This could be improved if it's a real problem in practice.

`Other Forms of MTask Balancing`

The largest source of runtime overhead is idle CPUs, which happens due to variance between our predicted runtime for each MTask and its actual runtime. That variance is magnified if MTasks are homogeneous, containing similar repeating logic which was generally close together in source code and which is still packed together even after going through Verilator's digestive tract.

If Verilator could avoid doing that, and instead would take source logic that was close together and distribute it across MTasks, that would increase the diversity of any given MTask, and this should reduce variance in the cost estimates.

One way to do that might be to make various "tie breaker" comparison routines in the sources to rely more heavily on randomness, and generally try harder not to keep input nodes together when we have the option to scramble things.

`Performance Regression`

It would be nice if we had a regression of large designs, with some diversity of design styles, to test on both single- and multi-threaded modes. This would help to avoid performance regressions, and also to evaluate the optimizations while minimizing the impact of parasitic noise.

`Per-Instance Classes`

If we have multiple instances of the same module, and they partition differently (likely; we make no attempt to partition them the same) then the variable sort will be suboptimal for either instance. A possible improvement would be to emit a unique class for each instance of a module, and sort its variables optimally for that instance's code stream.

Verilated Flow

The evaluation loop outputted by Verilator is designed to allow a single function to perform evaluation under most situations.

On the first evaluation, the Verilated code calls initial blocks, and then "settles" the modules, by evaluating functions (from always statements) until all signals are stable.

On other evaluations, the Verilated code detects what input signals have changes. If any are clocks, it calls the appropriate sequential functions (from always @ posedge statements). Interspersed with sequential functions it calls combo functions (from always @*). After this is complete, it detects any changes due to combo loops or internally generated clocks, and if one is found must reevaluate the model again.

For SystemC code, the eval() function is wrapped in a SystemC SC_METHOD, sensitive to all inputs. (Ideally it would only be sensitive to clocks and combo inputs, but tracing requires all signals to cause evaluation, and the performance difference is small.)

If tracing is enabled, a callback examines all variables in the design for changes, and writes the trace for each change. To accelerate this process the evaluation process records a bitmask of variables that might have changed; if clear, checking those signals for changes may be skipped.

4 CODING CONVENTIONS

Indentation style

To match the indentation of Verilator C++ sources, use 4 spaces per level, and leave tabs at 8 columns, so every other indent level is a tab stop.

All files should contain the magic header to insure standard indentation:

```
// -*- mode: C++; c-file-style: "cc-mode" -*-
```

This sets indentation to the cc-mode defaults. (Verilator predates a CC-mode change of several years ago which overrides the defaults with GNU style indentation; the c-set-style undoes that.)

The `astgen` script

Some of the code implementing passes is extremely repetitive, and must be implemented for each sub-class of `AstNode`. However, while repetitive, there is more variability than can be handled in C++ macros.

In Verilator this is implemented by using a Perl script, `astgen` to pre-process the C++ code. For example in `V3Const.cpp` this is used to implement the `visit()` functions for each binary operation using the TREEOP macro.

The original C++ source code is transformed into C++ code in the `obj_opt` and `obj_dbg` sub-directories (the former for the optimized version of Verilator, the latter for the debug version). So for example `V3Const.cpp` into `V3Const__gen.cpp`.

Visitor Functions

Verilator uses the *Visitor* design pattern to implement its refinement and optimization passes. This allows separation of the pass algorithm from the AST on which it operates. Wikipedia provides an introduction to the concept at http://en.wikipedia.org/wiki/Visitor_pattern.

As noted above, all visitors are derived classes of `AstNVisitor`. All derived classes of `AstNode` implement the `accept` method, which takes as argument a reference to an instance or a `AstNVisitor` derived class and applies the visit method of the `AstNVisitor` to the invoking AstNode instance (i.e. `this`).

One possible difficulty is that a call to `accept` may perform an edit which destroys the node it receives as argument. The `acceptSubtreeReturnEdits` method of `AstNode` is provided to apply `accept` and return the resulting node, even if the original node is destroyed (if it is not destroyed it will just return the original node).

The behavior of the visitor classes is achieved by overloading the `visit` function for the different `AstNode` derived classes. If a specific implementation is not found, the system will look in turn for overloaded implementations up the inheritance hierarchy. For example calling `accept` on `AstIf` will look in turn for:

```
void visit (AstIf* nodep)
void visit (AstNodeIf* nodep)
void visit (AstNodeStmt* nodep)
void visit (AstNode* nodep)
```

There are three ways data is passed between visitor functions.

1. A visitor-class member variable. This is generally for passing "parent" information down to children. `m_modp` is a common example. It's set to NULL in the constructor, where that node (`AstModule` visitor) sets it, then the children are iterated, then it's cleared. Children under an `AstModule` will see it set, while nodes elsewhere will see it clear. If there can be nested items (for example an `AstFor` under an `AstFor`) the variable needs to be save-set-restored in the `AstFor` visitor, otherwise exiting the lower for will lose the upper for's setting.

2. User attributes. Each `AstNode` (**Note.** The AST node, not the visitor) has five user attributes, which may be accessed as an integer using the `user1()` through `user5()` methods, or as a pointer (of type `AstNUser`) using the `user1p()` through `user5p()` methods (a common technique lifted from graph traversal packages).

   A visitor first clears the one it wants to use by calling `AstNode::user#ClearTree()`, then it can mark any node's user() with whatever data it wants. Readers just call `nodep->user()`, but may need to cast appropriately, so you'll often see `VN_CAST(nodep->userp(), SOMETYPE)`. At the top of each visitor are comments describing how the `user()` stuff applies to that visitor class. For example:

   ```
   // NODE STATE
   // Cleared entire netlist
   //   AstModule::user1p()     // bool. True to inline this module
   ```

   This says that at the `AstNetlist` `user1ClearTree()` is called. Each `AstModule`'s `user1()` is used to indicate if we're going to inline it.

   These comments are important to make sure a `user#()` on a given `AstNode` type is never being used for two different purposes.

   Note that calling `user#ClearTree` is fast, it doesn't walk the tree, so it's ok to call fairly often. For example, it's commonly called on every module.

3. Parameters can be passed between the visitors in close to the "normal" function caller to callee way. This is the second `vup` parameter of type `AstNUser` that is ignored on most of the visitor functions. V3Width does this, but it proved more messy than the above and is deprecated. (V3Width was nearly the first module written. Someday this scheme may be removed, as it slows the program down to have to pass vup everywhere.)

Iterators

`AstNVisitor` provides a set of iterators to facilitate walking over the tree. Each operates on the current `AstNVisitor` class (as this) and takes an argument type `AstNode*`.

`iterate`
: This just applies the `accept` method of the `AstNode` to the visitor function.

`iterateAndNextIgnoreEdit`
: Applies the `accept` method of each `AstNode` in a list (i.e. connected by `nextp` and `backp` pointers).

`iterateAndNext`
: Applies the `accept` method of each `AstNode` in a list. If a node is edited by the call to `accept`, apply `accept` again, until the node does not change.

`iterateListBackwards`
: Applies the `accept` method of each `AstNode` in a list, starting with the last one.

`iterateChildren`
: Apply the `iterateAndNext` method on each child `op1p` through `op4p` in turn.

`iterateChildrenBackwards`
: Apply the `iterateListBackwards` method on each child `op1p` through `op4p` in turn.

Caution on Using Iterators When Child Changes

Visitors often replace one node with another node; V3Width and V3Const are major examples. A visitor which is the parent of such a replacement needs to be aware that calling iteration may cause the children to change. For example:

```
// nodep->lhsp() is 0x1234000
iterateAndNextNull(nodep->lhsp());  // and under covers nodep->lhsp() changes
// nodep->lhsp() is 0x5678400
iterateAndNextNull(nodep->lhsp());
```

Will work fine, as even if the first iterate causes a new node to take the place of the lhsp(), that edit will update nodep->lhsp() and the second call will correctly see the change. Alternatively:

```
lp = nodep->lhsp();
// nodep->lhsp() is 0x1234000, lp is 0x1234000
iterateAndNextNull(lp); **lhsp=NULL;**  // and under covers nodep->lhsp() changes
// nodep->lhsp() is 0x5678400, lp is 0x1234000
iterateAndNextNull(lp);
```

This will cause bugs or a core dump, as lp is a dangling pointer. Thus it is advisable to set lhsp=NULL shown in the *'s above to make sure these dangles are avoided. Another alternative used in special cases mostly in V3Width is to use acceptSubtreeReturnEdits, which operates on a single node and returns the new pointer if any. Note acceptSubtreeReturnEdits does not follow nextp() links.

```
lp = acceptSubtreeReturnEdits(lp)
```

Identifying derived classes

A common requirement is to identify the specific `AstNode` class we are dealing with. For example a visitor might not implement separate `visit` methods for `AstIf` and `AstGenIf`, but just a single method for the base class:

```
void visit (AstNodeIf* nodep)
```

However that method might want to specify additional code if it is called for `AstGenIf`. Verilator does this by providing a `VN_CAST` method for each possible node type, using C++ `dynamic_cast`. This either returns a pointer to the object cast to that type (if it is of class `SOMETYPE`, or a derived class of `SOMETYPE`) or else NULL. So our `visit` method could use:

```
if (VN_CAST(nodep, AstGenIf) {
    <code specific to AstGenIf>
}
```

A common test is for `AstNetlist`, which is the node at the root of the AST.

5 TESTING

For an overview of how to write a test see the BUGS section of the Verilator primary manual.

It is important to add tests for failures as well as success (for example to check that an error message is correctly triggered).

Tests that fail should by convention have the suffix `_bad` in their name, and include `fails => 1` in either their `compile` or `execute` step as appropriate.

Preparing to Run Tests

For all tests to pass you must install the following packages:

* SystemC to compile the SystemC outputs, see http://systemc.org

* Parallel::Forker from CPAN to run tests in parallel, you can install this with e.g. "sudo cpan install Parallel::Forker".

* vcddiff to find differences in VCD outputs. See the readme at https://github.com/veripool/vcddiff

Controlling the Test Driver

Test drivers are written in PERL. All invoke the main test driver script, which can provide detailed help on all the features available when writing a test driver.

```
test_regress/t/driver.pl --help
```

For convenience, a summary of the most commonly used features is provided here. All drivers require a call to `compile` subroutine to compile the test. For run-time tests, this is followed by a call to the `execute` subroutine. Both of these functions can optionally be provided with a hash table as argument specifying additional options.

The test driver assumes by default that the source Verilog file name matches the PERL driver name. So a test whose driver is `t/t_mytest.pl` will expect a Verilog source file `t/t_mytest.v`. This can be changed using the `top_filename` subroutine, for example

```
top_filename("t/t_myothertest.v");
```

By default all tests will run with major simulators (Icarus Verilog, NC, VCS, ModelSim) as well as Verilator, to allow results to be compared. However if you wish a test only to be used with Verilator, you can use the following:

```
$Self->{vlt} or $Self->skip("Verilator only test");
```

Of the many options that can be set through arguments to `compiler` and `execute`, the following are particularly useful:

`verilator_flags2`
: A list of flags to be passed to verilator when compiling.

`fails`
: Set to 1 to indicate that the compilation or execution is intended to fail.

For example the following would specify that compilation requires two defines and is expected to fail.

```
compile (
    verilator_flags2 => ["-DSMALL_CLOCK -DGATED_COMMENT"],
    fails => 1,
    );
```

Regression Testing for Developers

Developers will also want to call ./configure with two extra flags:

--enable-ccwarn

Causes the build to stop on warnings as well as errors. A good way to ensure no sloppy code gets added, however it can be painful when it comes to testing, since third party code used in the tests (e.g. SystemC) may not be warning free.

--enable-longtests

In addition to the standard C, SystemC examples, also run the tests in the `test_regress` directory when using *make test*. This is disabled by default as SystemC installation problems would otherwise falsely indicate a Verilator problem.

When enabling the long tests, some additional PERL modules are needed, which you can install using cpan.

```
cpan install Unix::Processors
```

There are some traps to avoid when running regression tests

- When checking the MANIFEST, the test will barf on unexpected code in the Verilator tree. So make sure to keep any such code outside the tree.
- Not all Linux systems install Perldoc by default. This is needed for the *--help* option to Verilator, and also for regression testing. This can be installed using cpan:

  ```
  cpan install Pod::Perldoc
  ```

  Many Linux systems also offer a standard package for this. Red Hat/Fedora/Centos offer *perl-Pod-Perldoc*, while Debian/Ubuntu/Linux Mint offer *perl-doc*.
- Running regression may exhaust resources on some Linux systems, particularly file handles and user processes. Increase these to respectively 16,384 and 4,096. The method of doing this is system dependent, but on Fedora Linux it would require editing the `/etc/security/limits.conf` file as root.

6 DEBUGGING

--debug

When you run with --debug there are two primary output file types placed into the obj_dir, .tree and .dot files.

.dot output

Dot files are dumps of internal graphs in Graphviz http://www.graphviz.org/ dot format. When a dot file is dumped, Verilator will also print a line on stdout that can be used to format the output, for example:

```
dot -Tps -o ~/a.ps obj_dir/Vtop_foo.dot
```

You can then print a.ps. You may prefer gif format, which doesn't get scaled so can be more useful with large graphs.

For dynamic graph viewing consider ZGRViewer http://zvtm.sourceforge.net/zgrviewer.html. If you know of better viewers let us know; ZGRViewer isn't great for large graphs.

.tree output

Tree files are dumps of the AST Tree and are produced between every major algorithmic stage. An example:

```
    NETLIST 0x90fb00 <e1> {a0}
   1: MODULE 0x912b20 <e8822> {a8}  top  L2 [P]
  *1:2: VAR 0x91a780 <e74#> {a22} @dt=0xa2e640(w32)  out_wide [O] WIRE
   1:2:1: BASICDTYPE 0xa2e640 <e2149> {e24} @dt=this(sw32)  integer kwd=integer range=[31:0]
```

The following summarizes the above example dump, with more detail on each field in the section below.

"1:2:" indicates the hierarchy of the `VAR` is the `op2p` pointer under the `MODULE`, which in turn is the `op1p` pointer under the `NETLIST`

"VAR" is the AstNodeType.

"0x91a780" is the address of this node.

"<e74>" means the 74th edit to the netlist was the last modification to this node.

"{a22}" indicates this node is related to line 22 in the source filename "a", where "a" is the first file read, "z" the 26th, and "aa" the 27th.

"@dt=0x..." indicates the address of the data type this node contains.

"w32" indicates the width is 32 bits.

"out_wide" is the name of the node, in this case the name of the variable.

"[O]" are flags which vary with the type of node, in this case it means the variable is an output.

In more detail the following fields are dumped common to all nodes. They are produced by the `AstNode::dump()` method:

Tree Hierarchy

The dump lines begin with numbers and colons to indicate the child node hierarchy. As noted above in Key Classes Used in the Verilator Flow, `AstNode` has lists of items at the same level in the AST, connected by the `nextp()` and `prevp()` pointers. These appear as nodes at the same level. For example after inlining:

```
 NETLIST 0x929c1c8 <e1> {a0} w0
1: MODULE 0x92bac80 <e3144> {e14} w0  TOP_t  L1 [P]
1:1: CELLINLINE 0x92bab18 <e3686#> {e14} w0  v -> t
1:1: CELLINLINE 0x92bc1d8 <e3688#> {e24} w0  v__DOT__i_test_gen -> test_gen
...
1: MODULE 0x92b9bb0 <e503> {e47} w0  test_gen  L3
...
```

AstNode type

The textual name of this node AST type (always in capitals). Many of these correspond directly to Verilog entities (for example `MODULE` and `TASK`), but others are internal to Verialtor (for example `NETLIST` and `BASICDTYPE`).

Address of the node

A hexadecimal address of the node in memory. Useful for examining with the debugger.

Last edit number

Of the form <ennnn> or <ennnn#> , where nnnn is the number of the last edit to modify this node. The trailing # indicates the node has been edited since the last tree dump (which typically means in the last refinement or optimization pass). GDB can watch for this, see Debugging with GDB.

Source file and line

Of the form {xxnnnn}, where C{xx} is the filename letter (or letters) and nnnn is the line number within that file. The first file is a, the 26th is z, the 27th is aa and so on.

User pointers

Shows the value of the node's user1p...user5p, if non-NULL.

Data type

Many nodes have an explicit data type. "@dt=0x..." indicates the address of the data type (AstNodeDType) this node uses.

If a data type is present and is numeric, it then prints the width of the item. This field is a sequence of flag characters and width data as follows:

`s` if the node is signed.

`d` if the node is a double (i.e a floating point entity).

`w` always present, indicating this is the width field.

`u` if the node is unsized.

`/nnnn` if the node is unsized, where `nnnn` is the minimum width.

Name of the entity represented by the node if it exists

For example for a `VAR` it is the name of the variable.

Many nodes follow these fields with additional node specific information. Thus the `VARREF` node will print either `[LV]` or `[RV]` to indicate a left value or right value, followed by the node of the variable being referred to. For example:

```
1:2:1:1: VARREF 0x92c2598 <e509> {e24} w0  clk [RV] <- VAR 0x92a2e90 <e79> {e18} w0  clk [
```

In general, examine the `dump()` method in `V3AstNodes.cpp` of the node type in question to determine additional fields that may be printed.

The `MODULE` has a list of `CELLINLINE` nodes referred to by its `op1p()` pointer, connected by `nextp()` and `prevp()` pointers.

Similarly the `NETLIST` has a list of modules referred to by its `op1p()` pointer.

Debugging with GDB

The test_regress/driver.pl script accepts --debug --gdb to start Verilator under gdb and break when an error is hit or the program is about to exit. You can also use --debug --gdbbt to just backtrace and then exit gdb. To debug the Verilated executable, use --gdbsim.

If you wish to start Verilator under GDB (or another debugger), then you can use --debug and look at the underlying invocation of verilator_dgb. For example

```
t/t_alw_dly.pl --debug
```

shows it invokes the command:

```
../verilator_bin_dbg --prefix Vt_alw_dly --x-assign unique --debug
  -cc -Mdir obj_dir/t_alw_dly --debug-check -f input.vc t/t_alw_dly.v
```

Start GDB, then `start` with the remaining arguments.

```
gdb ../verilator_bin_dbg
...
(gdb) start --prefix Vt_alw_dly --x-assign unique --debug -cc -Mdir
          obj_dir/t_alw_dly --debug-check  -f input.vc t/t_alw_dly.v
          > obj_dir/t_alw_dly/vlt_compile.log
...
Temporary breakpoint 1, main (argc=13, argv=0xbfffefa4, env=0xbfffefdc)
    at ../Verilator.cpp:615
615         ios::sync_with_stdio();
(gdb)
```

You can then continue execution with breakpoints as required.

To break at a specific edit number which changed a node (presumably to find what made a <e####> line in the tree dumps):

```
watch AstNode::s_editCntGbl==####
```

To print a node:

```
pn nodep
# or: call nodep->dumpGdb() # aliased to "pn" in src/.gdbinit
pnt nodep
# or: call nodep->dumpTreeGdb()  # aliased to "pnt" in src/.gdbinit
```

When GDB halts, it is useful to understand that the backtrace will commonly show the iterator functions between each invocation of `visit` in the backtrace. You will typically see a frame sequence something like

```
...
visit()
iterateChildren()
iterateAndNext()
accept()
visit()
...
```

7 ADDING A NEW FEATURE

Generally what would you do to add a new feature?

1. File a bug (if there isn't already) so others know what you're working on.
2. Make a testcase in the test_regress/t/t_EXAMPLE format, see TESTING.
3. If grammar changes are needed, look at the git version of VerilogPerl's src/VParseGrammar.y, as this grammar supports the full SystemVerilog language and has a lot of back-and-forth with Verilator's grammar. Copy the appropriate rules to src/verilog.y and modify the productions.
4. If a new Ast type is needed, add it to V3AstNodes.h.

Now you can run "test_regress/t/t_{new testcase}.pl --debug" and it'll probably fail but you'll see a test_regress/obj_dir/t_{newtestcase}/*.tree file which you can examine to see if the parsing worked. See also the sections above on debugging.

Modify the later visitor functions to process the new feature as needed.

Adding a new pass

For more substantial changes you may need to add a new pass. The simplest way to do this is to copy the `.cpp` and `.h` files from an existing pass. You'll need to add a call into your pass from the `process()` function in `src/verilator.cpp`.

To get your pass to build you'll need to add its binary filename to the list in `src/Makefile_obj.in` and reconfigure.

8 DISTRIBUTION

The latest version is available from http://www.veripool.org/.