Verilator 3.840 Internals Manual

Wilson Snyder
http://www.veripool.org

2012-07-31

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 ADDING A NEW FEATURE

Generally what would you do to add a new feature?

File a bug (if there isn't already) so others know what you're working on.

Make a testcase in the test_regress/t/t_EXAMPLE format, see TESTING Below.

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.

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 below on debugging.

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

4 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 psudo-flattened. Each module's variables and functions get "Scope" references. A scope reference is an occurrence of that un-flattened 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 psudo-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

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 `op1p` to give the pointer to the AST for the "then" block, while `elsesp` calls `op2p` 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.

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.

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.

5 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

The 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, AstNUser* vup)
void visit (AstNodeIf* nodep, AstNUser* vup)
void visit (AstNodeStmt* nodep, AstNUser* vup)
void visit (AstNode* nodep, AstNUser* vup)
```

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 `nodep->userp()->castSOMETYPE()`. 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

`AstNode` provides a set of iterators to facilitate walking over the tree. Each takes two arguments, a visitor, `v`, of type `AstNVisitor` and an optional pointer user data, `vup`, of type `AstNuser*`. The second is one of the ways to pass parameters to visitors described in Visitor Functions, but its use is no deprecated and should be used for new visitor classes.

`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.

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, AstNUser* vup)
```

However that method might want to specify additional code if it is called for `AstGenIf`. Verilator does this by providing a `castSOMETYPE()` 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 (nodep->castAstGenIf()) {
    <code specific to AstGenIf>
}
```

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

6 TESTING

To write a test see the BUGS section of the Verilator primary manual, and the documentation in:

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

7 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]
```

"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 below.

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 numberic, it then prints the width of the item. This field is a squence 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()
...
```

8 DISTRIBUTION

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