VERILATOR

Verilator Internals #1: Debug, Stages, AstNode, V3Graph

Wilson Snyder, 2020

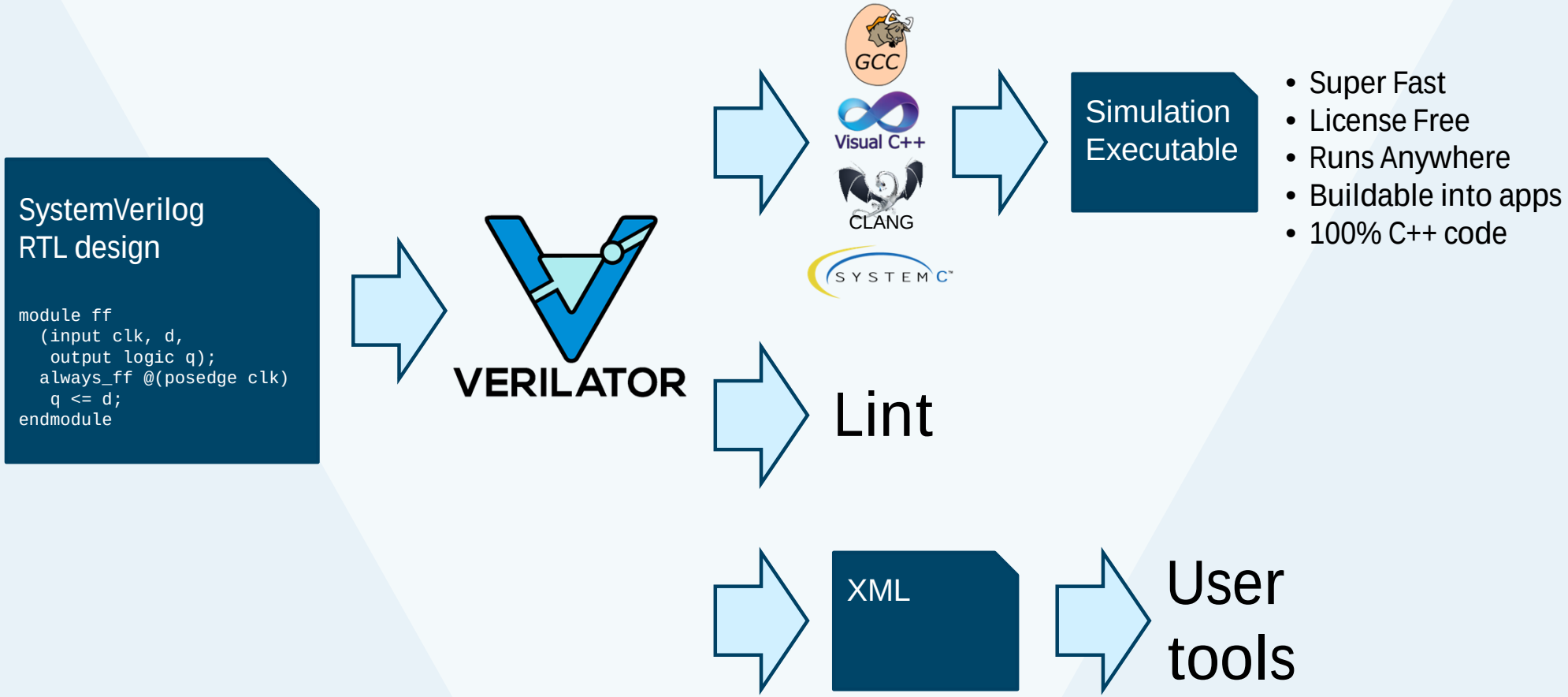
Agenda

This is the first in a series of talks on the internals of Verilator.

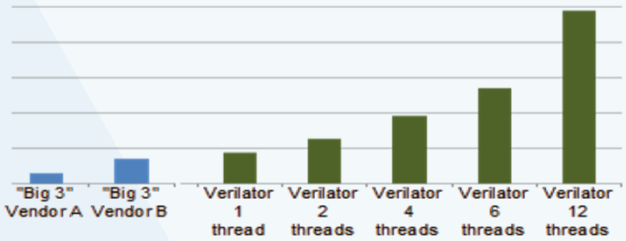
- Overview
- Compile Phases
- AstNode
- V3Graph
- Debug

Please see docs/internals.adoc for additional information.

Verilator


SystemVerilog
RTL design
module ff
(input clk, d,
output logic q);
always_ff @(posedge clk)
q <= d;
endmodule
VERILATOR
GCC
Visual C++
CLANG
SYSTEMC
Simulation
Executable
Super Fast
License Free
Runs Anywhere
Buildable into apps
100% C++ code
Lint
XML
User
tools

Why Verilator?



Fast

- Outperforms many commercial simulators
- Single- and multi-threaded output models

Widely Used

- Wide industry and academic deployment
- Out-of-the-box support from Arm, and RISC-V vendor IP

Community Driven, Professionally Supported, & Openly Licensed

- Guided by the CHIPS Alliance and Linux Foundation
- Professional support available through partners
- Open, and free as in both speech and beer
- More simulation for your verification budget

Verilator Phases

Compile Phases (1 of 9: Elaboration)

- Link (V3Link*.cpp)
  - Resolve module and signal references
  - Calls Bison to parse design (V3Parse*.cpp)
    - Calls Flex to perform lexical analysis
    - Calls Verilog preprocessor (V3PreProc*.cpp)
- Parameterize (V3Param.cpp)
  - Propagate parameters, and make a unique module for each set of parameters
- Elaborate expression widths (V3Width*.cpp)
  - Elaboration and data type assignments (complicated)

Compile Phases (2 of 9: Module based)

- Coverage Insertion (V3Coverage.cpp)
  - Insert line coverage statements at all if, else, and case conditions
- Constification (V3Const.cpp) & dead code elimination (V3Dead.cpp)
  - Eliminate constants and simplify expressions
  - ("if (a) b=1; else b=c;" becomes "b = a|c";)
- Assertions (V3Assert.cpp)
  - Convert explicit and user assertions into standard sequential code

Compile Phases (3 of 9: Module based)

- Hierarchical link
  - Flatten begin/end statements (V3Begin.cpp)
  - Hierarchical link (V3LinkDot.cpp)
- Tristate elimination
  - Convert case statements into if statements (V3Tristate.cpp)
  - Replace assignments to X with random values. (V3Unknown.cpp)
- Inline and connect modules
  - Module inlining optimization (V3Inline.cpp)
  - Hierarchical link (V3LinkDot.cpp)
  - Connect non-inlined modules (V3Inst.cpp)

Compile Phases (4 of 9: Scope Based)

- Scope (V3Scope.cpp)
  - If a module is instantiated 5 times, make 5 copies of its variables
- Scope transformations
  - Move class statements to netlist scope (V3Class.cpp)
  - Convert case statements to normal statements (V3Case.cpp)
  - Attach or inline functions/tasks (V3Task.cpp)
  - Rename variables (V3Name.cpp)
  - Loop unrolling optimization (V3Unroll.cpp)
  - Replace slices with statements (V3Slice.cpp)

Compile Phases (5 of 9: Scope Based)

VERILATOR

- Scope transformations
  - Redundant assignment removal optimization (V3Life.cpp)
  - Statement tables optimization (V3Table.cpp)
  - Move always into by clock domain active blocks (V3Active.cpp)
  - Break always blocks into pieces to aid ordering (V3Split.cpp)
    - always @ ... begin a<=z; b<=a; end

Compile Phases (6 of 9: Scope Based)

- Gate optimization (V3Gate.cpp)
  - Treat the design as a netlist, and optimize away buffers, inverters, etc.
- Delayed assignment flattening (V3Delayed.cpp)
  - a<=a+1 becomes “a_dly=a; ….; a_dly=a+1; …. a=a_dly;”
- Ordering (V3Order.cpp)
  - Determine the execution order of each always/wire statement.
    - This may result in breaking combinatorial feedback paths.
    - The infamous “not optimizable” warning

Compile Phases (7 of 9: Scope Based)

- Clocking
  - Find generated clocks (V3GenClock.cpp)
  - Make master eval() (V3Clock.cpp)
- Lifetime analysis optimization (V3Life.cpp, V3LifePost.cpp)
  - If a variable is never used after being set, no need to set it.
  - If code ordering worked right, this eliminates _dly variables.
- Handle combo blocks (V3Changed.cpp)
  - If combo code loops, make a change_detect to reevaluate

Compile Phases (8 of 9: Backend)

- Descope (V3Descope.cpp)
  - Reverse the flattening process; work on C++ classes now.
- Localize & combine optimizations
  - Move variables from heap to stack when possible (V3Localize.cpp)
  - Identify duplicate functions and share code (V3Combine.cpp)
- Clean, premit, expand & substitute
  - Add AND statements to correct expression widths (V3Clean.cpp)
  - Create temporaries for wide ops (V3Premit.cpp)
  - Expand wide and other internal operations into C++ (V3Expand.cpp)
  - Substitute temporaries with the value of the temporary (V3Substitute.cpp)

Compile Phases (9 of 9: Backend)

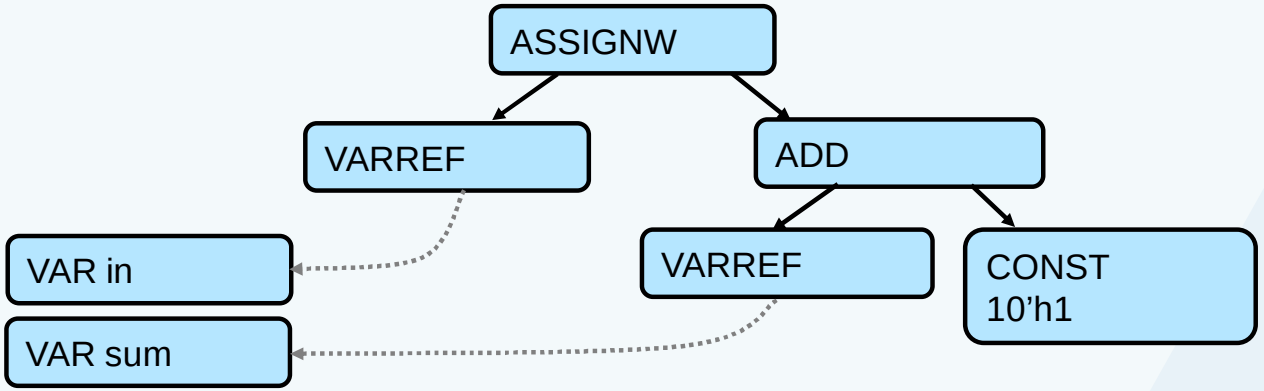
- Cache optimizations
  - Merge ? : assignments into if statements (V3MergeCond.cpp)
  - Convert flattened assignments into array back into a loop (V3Reloop.cpp)
  - Branch predict (V3Branch.cpp)
- C++ correctness fixups
  - Create casts to get proper math (V3Cast.cpp)
  - Create constructors/destructors (V3CCtors.cpp)
- Emit code (V3Emit*.cpp)

AstNodes

AstNode

The core internal structure is an AstNode

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



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

```
1:2: ASSIGNW 0xa097 <e1312> {e29} @dt=0x9b
1:2:1: ADD 0xa098 <e774> {e29} @dt=0x9b0@(w10)
1:2:1:1: VARREF 0xa099 <e781> {e29} @dt=0x9b0@(w10)  in [RV] <- VAR in
1:2:1:2: CONST 0xa09a <e1556> {e29} @dt=0x9b0@(w10)  10'h1
1:2:2: VARREF 0xa09c <e754> {e29} @dt=0x9b0@(w10)  sum [LV] => VAR sum
```

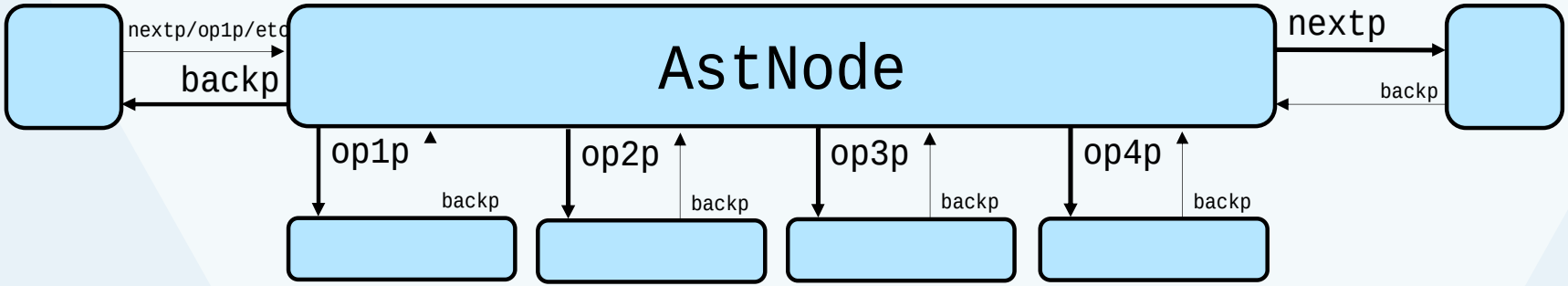
Source Code Line Number.

Data type (node address and simplified data type description)

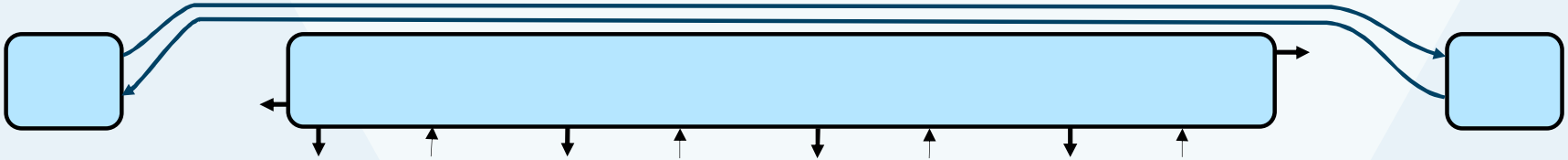
Node Address

LV indicates an lvalue – it sets the variable.

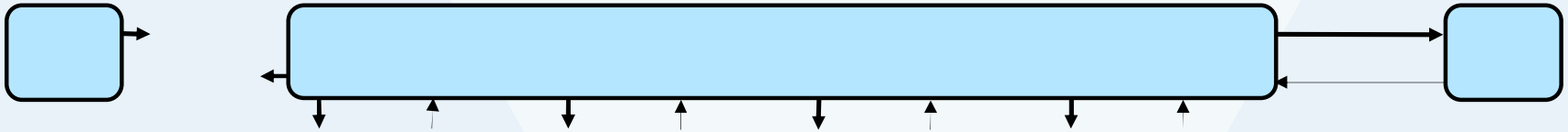
AstNode Pointers



`unlinkFrBack` detaches *backp* and makes *nodep* separate from parent



`unlinkFrBackWithNext` detaches *backp* and makes *nodep* and next(s) separate



Visitor Template

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

```
class TransformVisitor : AstNVisitor {

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

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 (i.e. op1p, op2p, op3p, op4p); 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.

userp

- user#p's track information about a node for algorithms
- Faster alternative to std::map with nodep as a key

```
class TransformVisitor : AstNVisitor {

  AstUser1InUse m_inuser1;

  virtual void visit(AstX* nodep) {
    gotp = nodep->userp1();
    nodep->user1p(ptrp);
  }
  virtual void visit(AstY* nodep) {
    AstNode::user1ClearTree();
  }
}
```

Effectively gotp = map[nodep]

Effectively map[nodep] = ptrp;

Globally clears every userp1(). This is an O(1) operation, takes only a few instructions. (Unlike if we were using a std::map.)

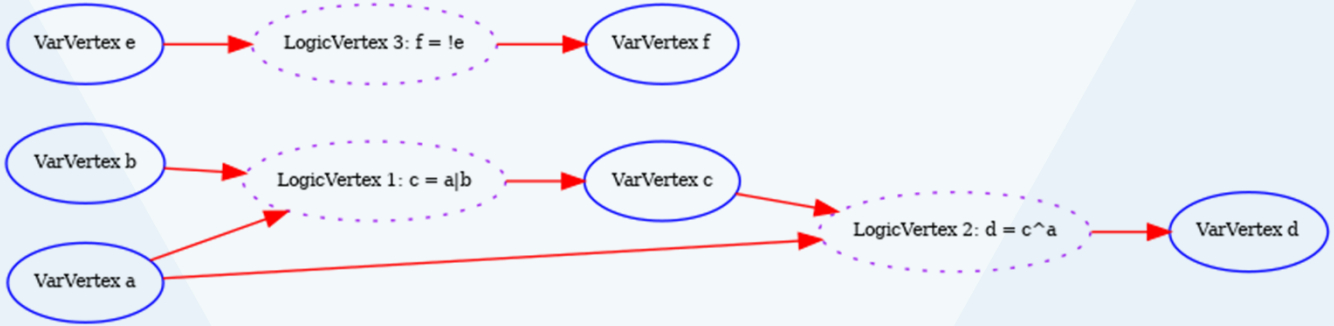
V3Graph

Directed Graphs

- Consider code:

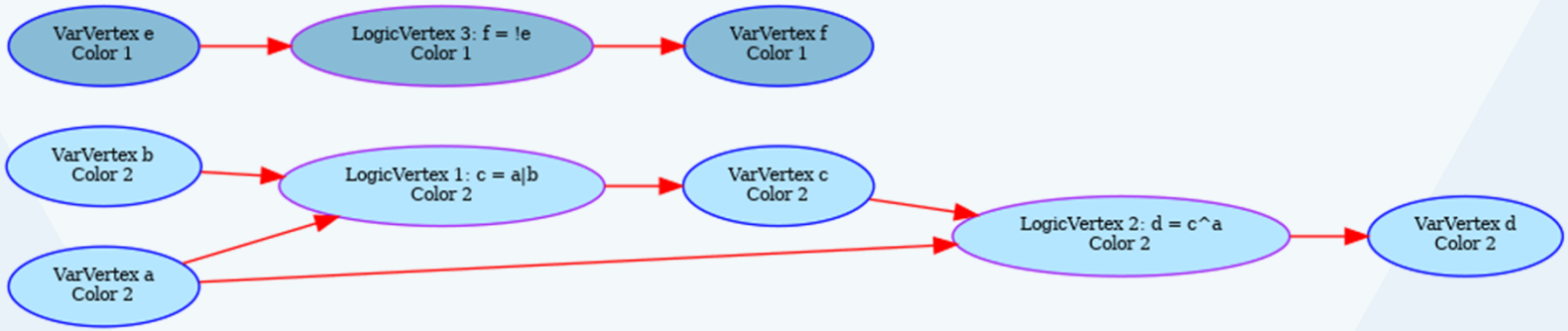
```
1: c = a | b;
2: d = c ^ a;
3: f = !e;
```

- We assign vertices to each variable (a, b, c, d, e, f) and to each statement (1, 2, 3). We then attach edges between each variable's vertex and each statement's vertex.

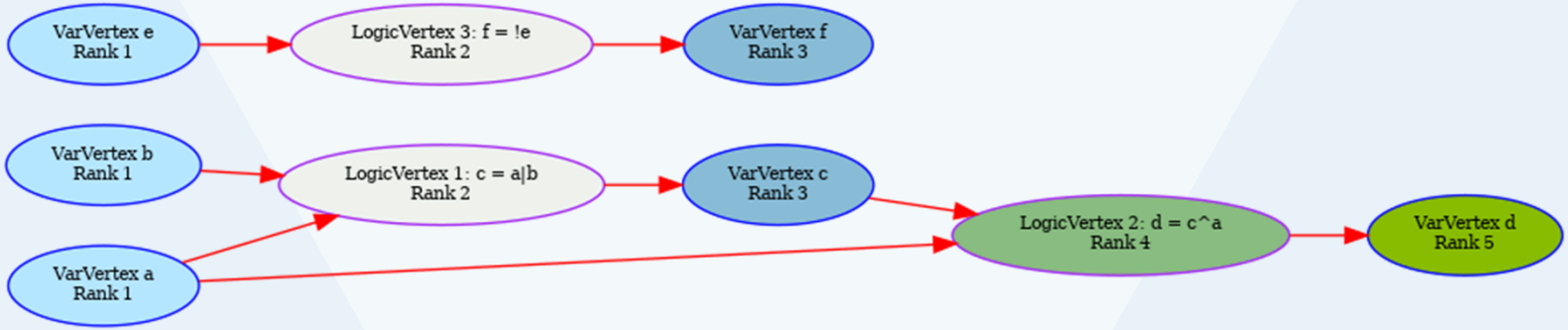


Directed Graph Algorithms

- Color – Each subgraph gets different `color()`



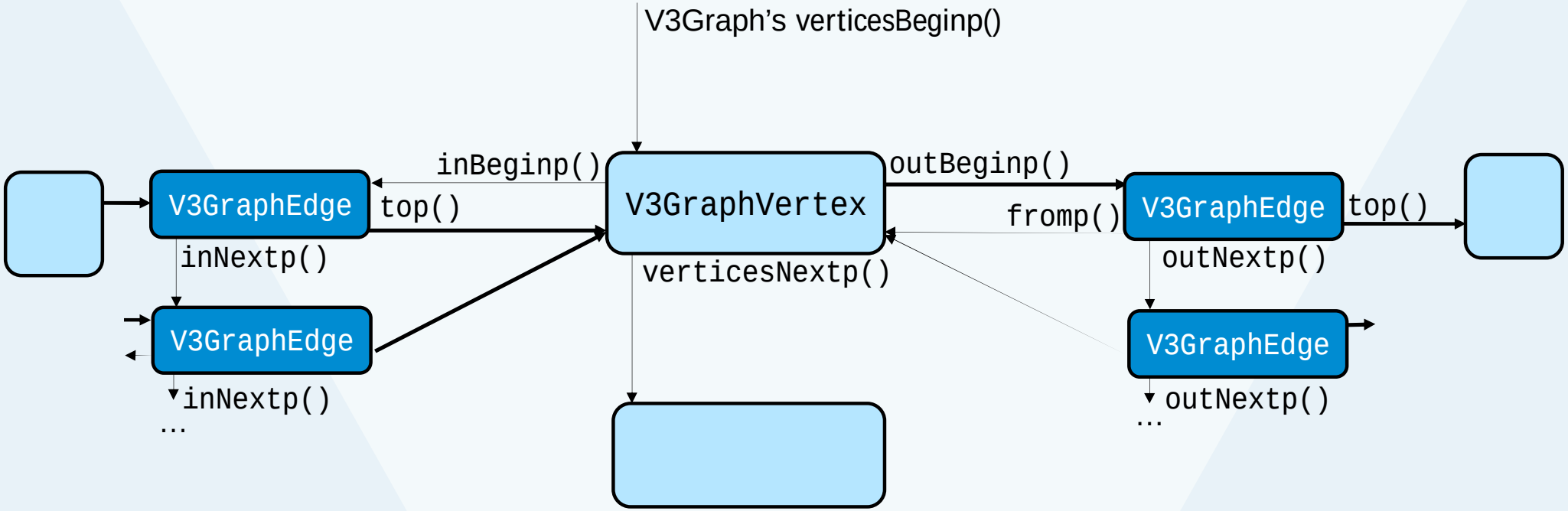
- Rank – Assign `rank()` based on depth from following edges



Graph Optimizations

- 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 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

V3Graph Structures



V3Graph Debugging

- When run with debug, message indicates command for .dot
  `dot -Tpdf -o` *`Vfile.pdf Vfile.dot`*

- GraphViz supports many other options, e.g.
  `dot -Tpng -o` *`Vile.png Vfile.dot`*

- Other viewers described in docs/internals.adoc

Verilator Debug

Debug Flags

`--debug`

- Use the debug executable
- Never reclaim node memory (to detect dangling pointers)
- Turn on `--debugi 3` (also enables creating some .dot files)
- Turn on `--dump-treei 3` (create .tree files)
- Turn on `--stats` (create stats.txt file)

`--debugi 9 --dump-treei 9`

- Turn on all UINFO's and also make more .tree files (generally too verbose)

`--debugi-`*`V3SourceFileName`* `9`

- Turn on UINFO for single source file (very useful when debugging one file)

`--no-dump-tree`

- Turn off .tree files (e.g. when they are getting too large)

Using GDB

- Common debug of a single file:

```
verilator … --debug –debugi-V3SourceFileName 9 –gdbbt
```

- May create backtrace if core dumps:

```
#9  0x# in BeginVisitor::visit (this=0x#, nodep=0x#) at V3Begin.cpp:245
#10 0x# in AstNVisitor::visit (this=0x#, nodep=0x#) at V3Ast__gen_visitor.h:229
#11 0x# in AstNVisitor::visit (this=0x#, nodep=0x#) at V3Ast__gen_visitor.h:160
#12 0x# in AstInitial::accept (this=0x#, v=...) at V3AstNodes.h:3190
#13 0x# in AstNode::iterateAndNext (this=0x#, v=...) at V3Ast.cpp:833
#14 0x# in AstNode::iterateChildren (this=0x#, v=...) at V3Ast.cpp:797
#15 0x# in AstNVisitor::iterateChildren (this=0x#, nodep=0x#) at V3Ast.h:2903
#16 0x# in BeginVisitor::visit (this=0x#, nodep=0x#) at V3Begin.cpp:77
#17 0x# in AstNVisitor::visit (this=0x#, nodep=0x#) at ./V3Ast__gen_visitor.h:194
#18 0x# in AstModule::accept (this=0x#, v=...) at V3AstNodes.h:2491
```

  - In this example abort was at V3Begin.cpp line 245. This code was iterating under an AstInitial which is underneath an AstModule.

- For interactive GDB, use “—gdb” instead of “—gdbbt” and see docs/internals.adoc for some hints


VERILATOR