# Verilator:
# Speedy Reference Models, Direct from RTL

http://www.veripool.org/papers

Wilson Snyder

Veripool.org

wsnyder@wsnyder.org

## Agenda

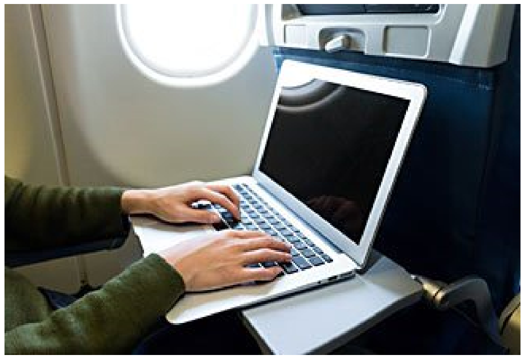
- Modeling Hardware
- Intro to Verilator
- Verilator Internals
- Futures
- Conclusion
- Q & A

## Modeling Hardware

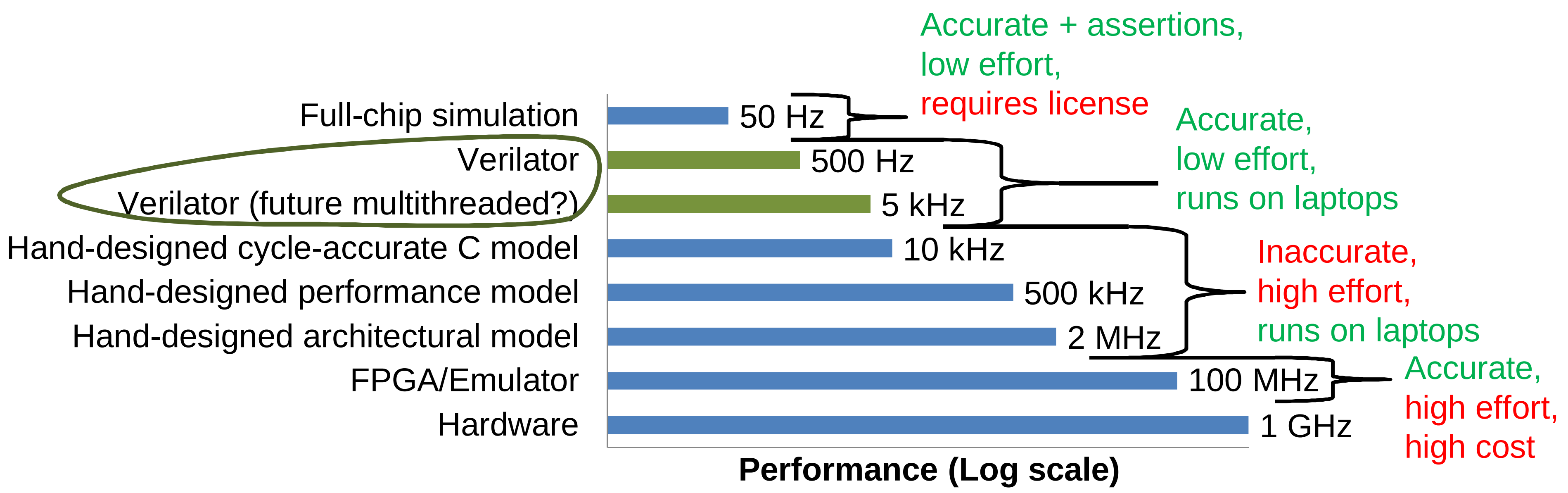
## Hardware/Software Co-design

- Products are a mix of hardware and software
- Therefore, both need to be designed in parallel
- What abstraction of HW can SW develop on?

- Ideal:
  - Accurate as real hardware
  - Fast as real hardware
  - Minimal effort to develop model
  - Free runtime – i.e. develop SW anywhere



## Levels of Abstraction vs. Performance



## Intro to Verilator

## Verilator is a Compiler

- Verilator compiles synthesizable Verilog into C++
  - Matches synthesis rules, not simulation rules
  - Time delays ignored (a <= #{n} b;)
  - Only two state simulation (and tri-state busses)
  - Unknowns are randomized (better than Xs)

- Creates C++/SystemC wrapper

- Creates own internal interconnect
  - Plays several tricks to get good, fast code

## Example: Getting Started

- (From https://www.veripool.org/projects/verilator/wiki/Installing)
- Install RPM

```
apt-get install verilator
```

- Or, the latest from sources

```
(once) git clone http://git.veripool.org/git/verilator
       cd verilator
       git pull
       autoconf
       ./configure
       make
       make install
```

- Read docs and example

```
verilator --help
```

## Example: First Module

Convert.v

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

   initial $display(“Hello flip-flop”);
   always_ff @ (posedge clk)
      out <= data;
endmodule
```

- Lint check your code

```
verilator –lint-only -Wall Convert.v
```

## Example: Translation

- Translate to a C++ class (also can do SystemC module – not shown)

```
verilator –cc Convert.v
```

Convert.v

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

   initial $display("Hello flip-flop");
   always_ff @ (posedge clk)
      out <= data;
endmodule
```

obj_dir/VConvert.h

```
#include "verilated.h"

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

   void eval();
   void final();
}
```

- Output in obj_dir/
- Verilog top module became a C++ class (obj_dir/V*Convert*.h)
- Inputs and outputs map directly to bool, uint32_t, or array of uint32_t's

## Example: Calling the model

- Write .cpp file to call the Verilated class
  - Verilator doesn't make time pass!
    - The key difference from verification-style simulators

top.cpp

```
#include "VConvert.h"

int main() {
    VConvert* topp = new VConvert();
    for (int time=0;
          time<100 && !Verilated::gotFinish()
          ++time) {
        topp->data = 1;
        topp->clk = !topp->clk;
        topp->eval();
        // … = topp->out();
    }
    topp->final();
}
```

obj_dir/VConvert.h

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

    void eval();
    void final();
}
```

## Example: Compile and run

- Compile with e.g. GCC

```
make –C obj_dir –f VConvert.mk VConvert
```

- Run

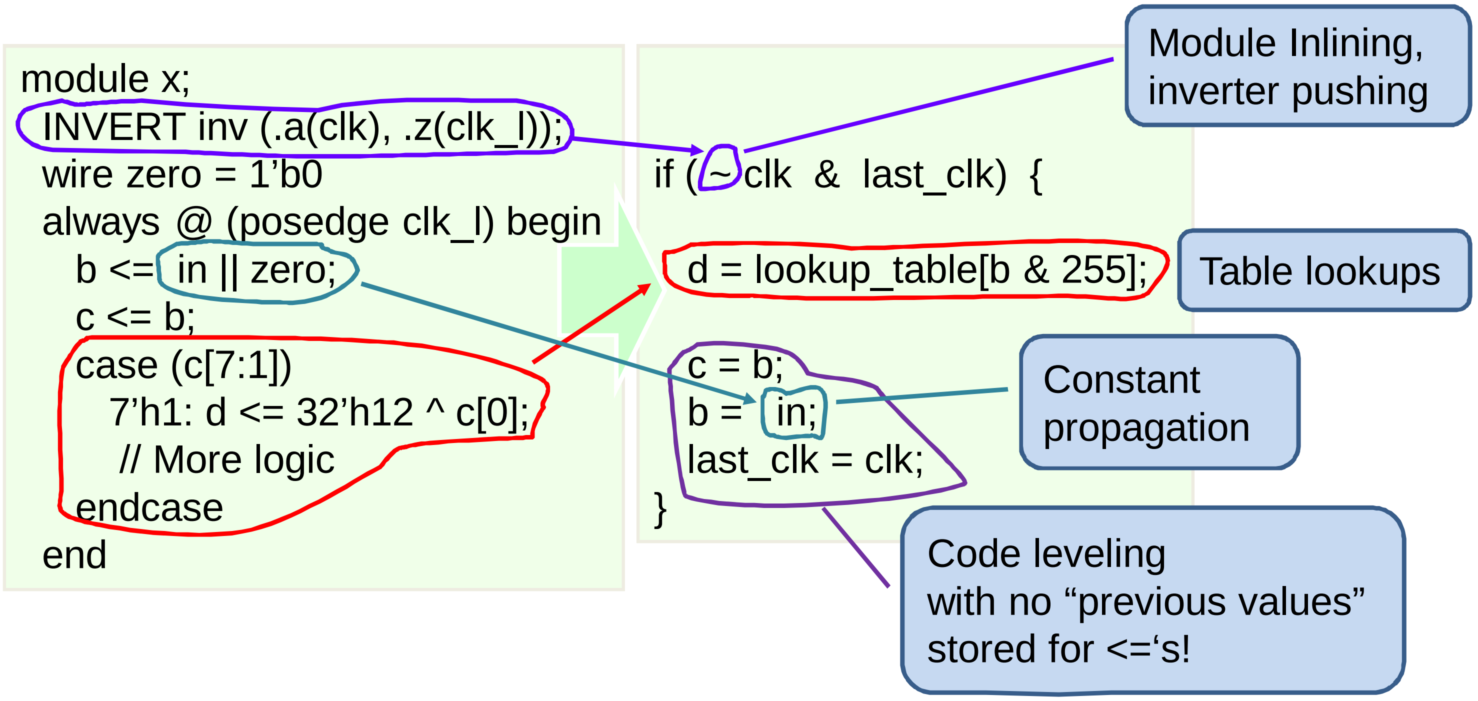
```
obj_dir/Vconvert
```

```
Hello flip-flop
```

- Other examples
  1. `verilator --help`
  2. "examples/" directory in the kit and installed files

## Verilator Internals

# Verilator Optimizations



```
module x;
  INVERT inv (.a(clk), .z(clk_l));
  wire zero = 1'b0
  always @ (posedge clk_l) begin
    b <= in || zero;
    c <= b;
    case (c[7:1])
      7'h1: d <= 32'h12 ^ c[0];
       // More logic
    endcase
  end
```

```
if (~clk  &  last_clk)  {

  d = lookup_table[b & 255];

  c = b;
  b = in;
  last_clk = clk;
}
```

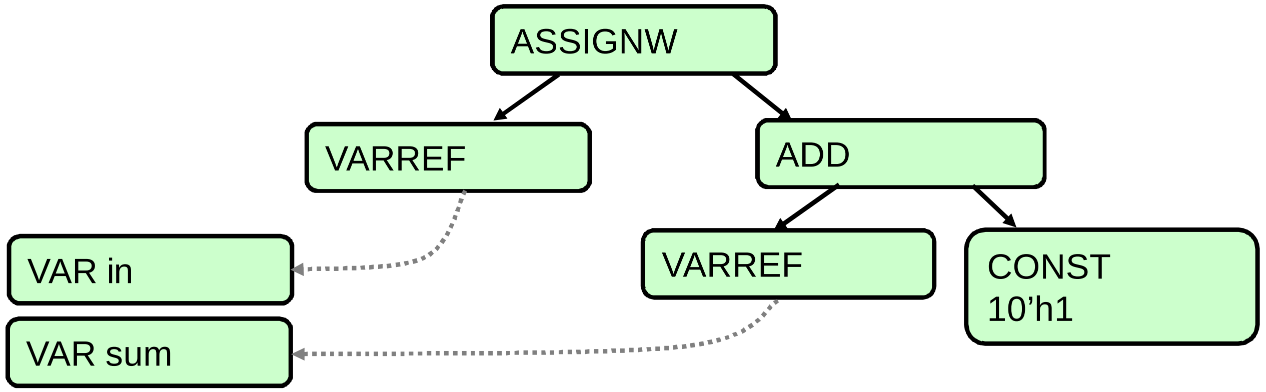
Module Inlining, inverter pushing
Table lookups
Constant propagation
Code leveling with no “previous values” stored for <=‘s!

- End result is extremely fast Verilog simulation

# 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=0x9
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

LV indicates an lvalue – it sets the variable.

VeriPool

## Futures

## Future Language Enhancements

- Generally, new features are added as requested, unless difficult ☺
  - Unpacked Structs, Classes and methods
  - Dynamic memory, new/delete
  - Event loop, fork/join
  - Someday, full UVM support?

- Lint
  - Improve Verilog code quality checks to aid designers in finding bugs

## Future Single-Threaded Performance

- Bit-splitting to avoid UNOPTFLAT
  - Bits in a vector that are always used separately should be separate signal

- Better merging of logic into parallel arrays

```
    wire a0 = b0 | c0; wire a1 = b1 | c1;
    -> wire a[1:0] = b[1:0] | c[1:0];
```

- Better icache packing by building up subroutines, structs and loops

```
    mod1.foo = some_long_equation (mod1.bar)
    mod0.foo = some_long_equation (mod0.bar])
    ->   for (i=0; i<2; ++i) mod[i].foo = function(mod[i].bar)
```

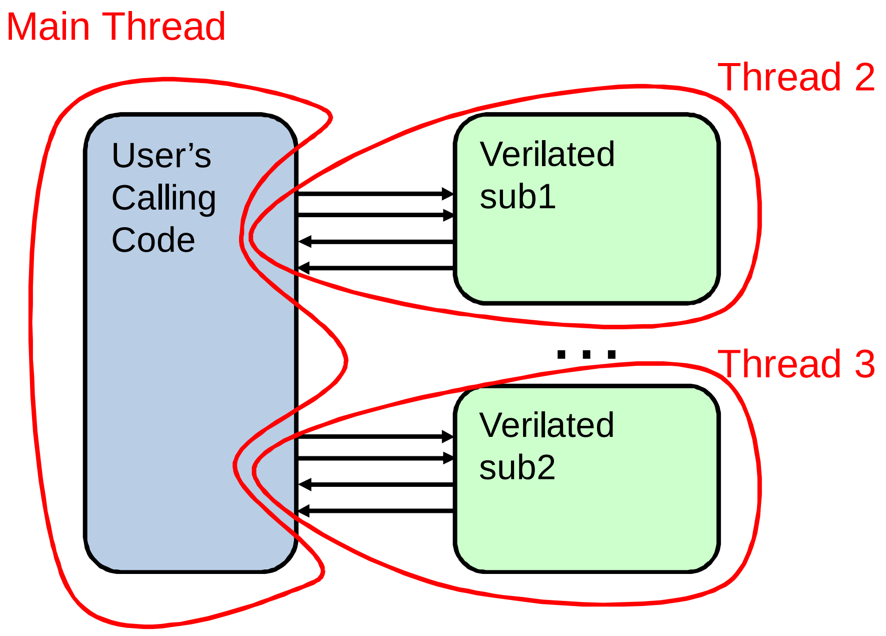
- Likewise dcache packing – Lots of room for research!

## Multithreaded Performance – The Easy Way

- Manually instantiate multiple Verilated models and user's wrapper threads them

top.cpp (PSEUDO CODE)

```
#include “Vsub1.h”
#include “Vsub2.h”
void thread1 {
    Vsub1* top1p = new Vsub1();
    barrier();  // Wait for thread1
    for (…) {
        …
        topp->eval();
        barrier(); // Align threads
    }
}
// thread 2 similar, using Vsub2
int main() {
    threads_create(thread1,thread2);
    wait(thread1,thread2);
}
```



sub1.v

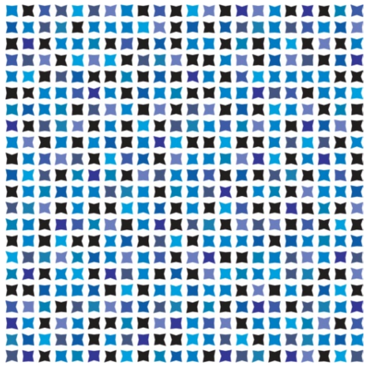
```
module sub1;
  …
endmodule
```

sub2.v

```
module sub2;
  …
endmodule
```

## Multithreaded Performance – Partitioning

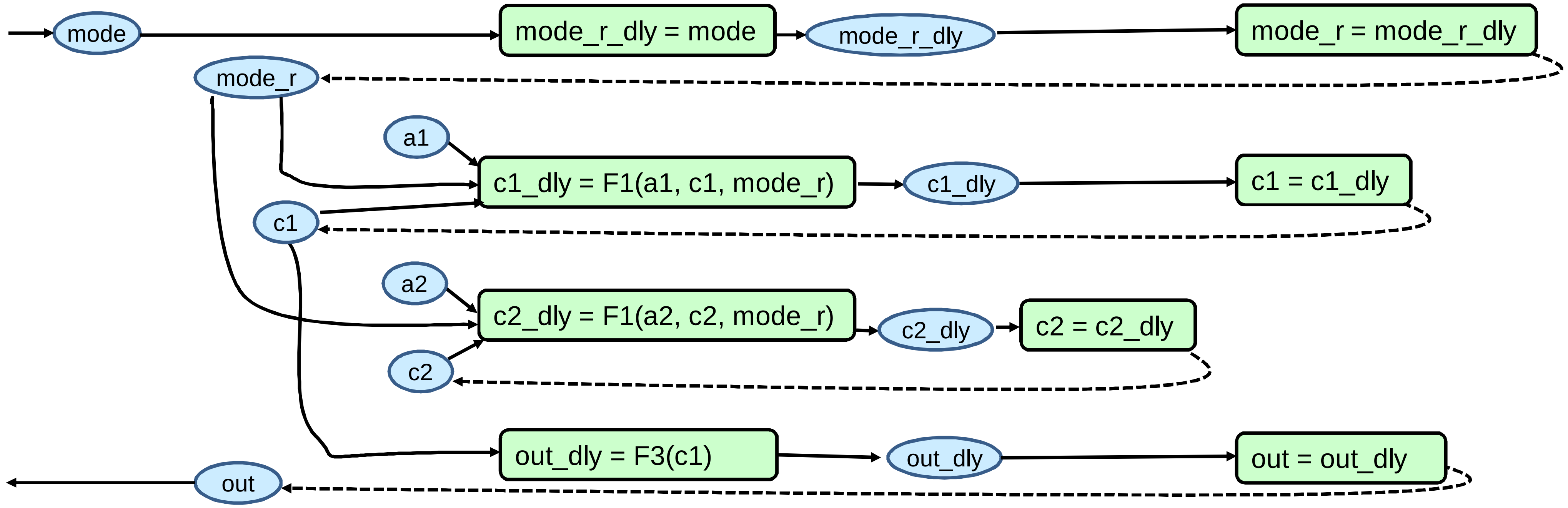
- True Automatic Model Partitioning
  - Decompose circuit scheduling graph into partitions
  - Schedule the partitions separately (“trains”)
  - Dynamically schedule trains on threads
  - Should look to users just like single-threaded



- Automatic parallelism of the whole socket!
  - ThunderX2 -> 32 cores, 128 threads, per socket!
  - Under best case can get superscalar performance if fits in caches
  - Great PhD thesis

# Locked Multithreading

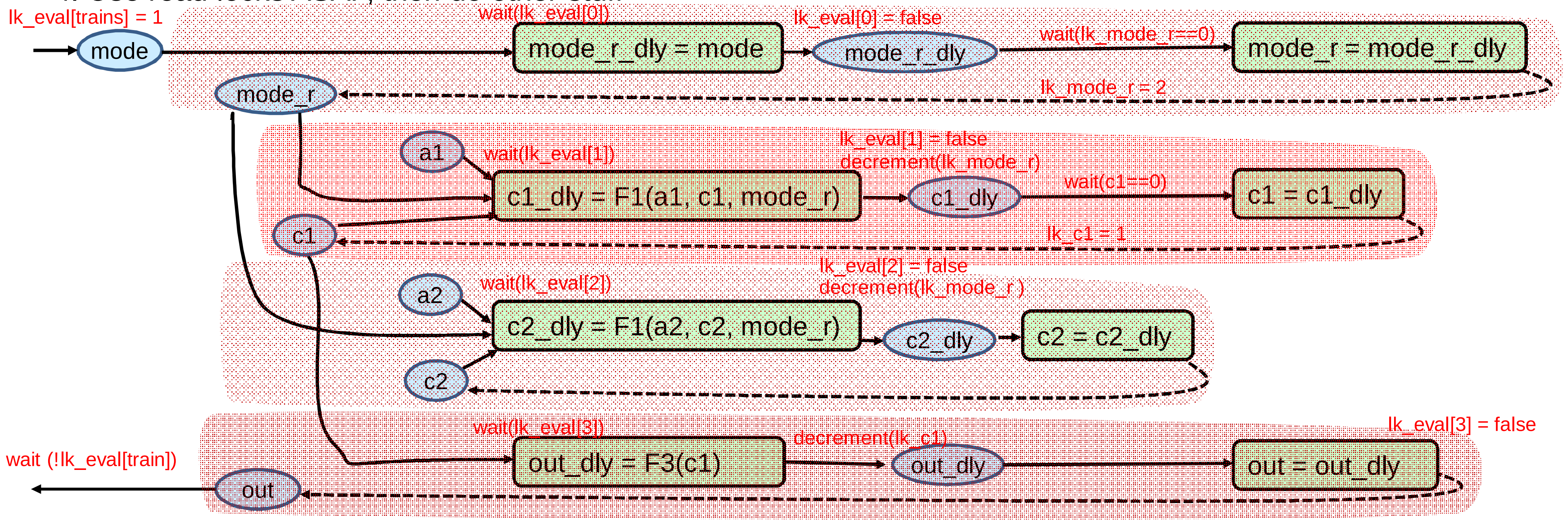
```
always_ff @ (posedge clk)
    mode_r <= mode;
    c1 <= F1(a1, c1, mode_r);
    c2 <= F2(a2, c2, mode_r);
    out <= F3(c1);
```



# Locked Multithreading

```
always_ff @ (posedge clk)
    mode_r <= mode;
    c1 <= F1(a1, c1, mode_r);
    c2 <= F2(a2, c2, mode_r);
    out <= F3(c1);
```

1. Graph is partitioned into “trains”.
2. Atomic locks picked to involve minimal signals
   (Probably a lock really covers multiple signals)
3. Compute locked outputs ASAP
4. Use read locks ASAP, then do other stuff



## Conclusion

## You Can Help (1 of 2)

- Firstly, need more testcases!
  - Many enhancements are gated by testing and debugging

- Large standalone test cases
  - Need a large testchips and simple testbenches.
  - ★ Port a large SoCs
  - Add a tracing and cycle-by-cycle shadow simulation mode, so finding introduced bugs is greatly simplified?

## You Can Help (2 of 2)

- Run gprof/oprofile and fix bottlenecks
  - Most optimizations came from “oh, this could be faster”
- Tell us what changes you’d like to see
  - We don’t hear from most users, and have no idea what they find frustrating.
- Advocate.
- Of course, patches and co-authors always wanted!

## Contributing Back

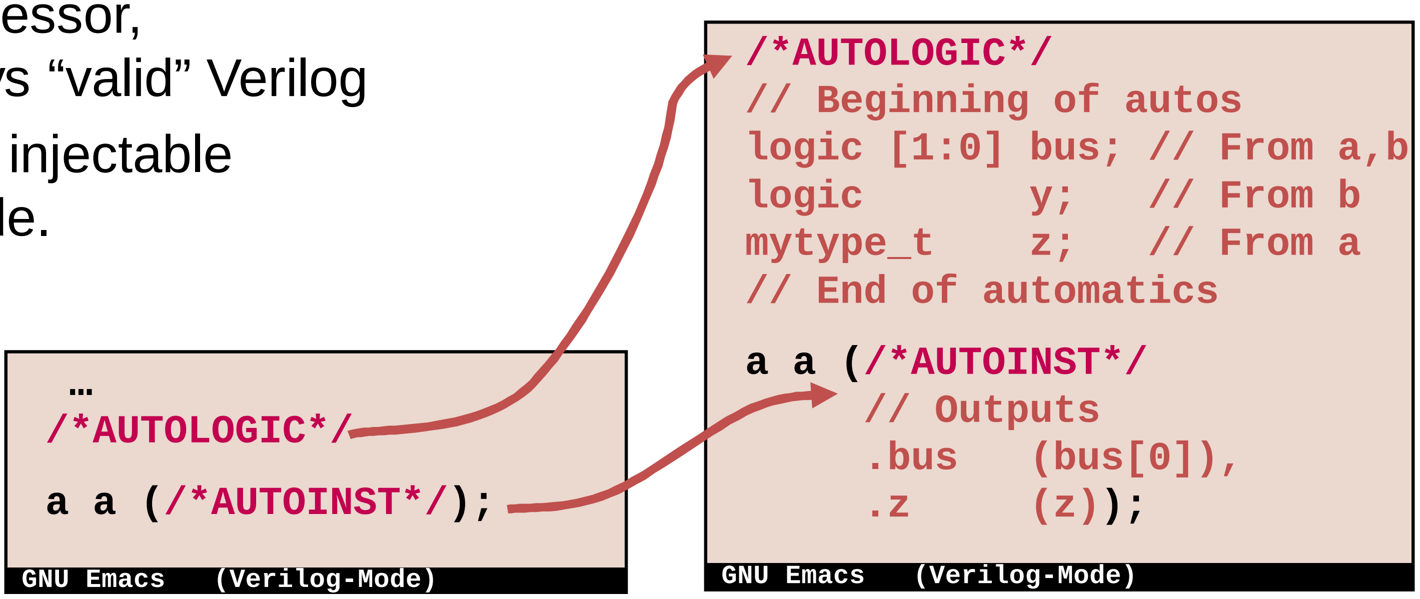
- The value of Open Source is in the Community!
- Use Forums
- Use Bug Reporting
  - Even if to say what changes you'd like to see
- Try to submit a patch yourself
  - Many problems take only a few hours to resolve yourself; often less time than packaging up a test case for an EDA company!
  - Even if just documentation fixes!
  - Great experience for the resume!
- Advocate

## Conclusions

- Adopt Verilator
  - Supported
    - Continual language improvements
    - Growing support network for 20+ years
    - Run faster than major simulators
  - Open Source Helps You
    - Easy to run on laptops or SW developer machines
    - Get bug fixes in minutes rather than months
    - Greatly aids commercial license negotiation
  - Keep your Commercial Simulators
    - SystemVerilog Verification, analog models, gate SDF, etc.

# Verilog-Mode for Emacs

- Thousands of users, including most IP houses
- Fewer lines of code to edit means fewer bugs
- Indents code correctly, too
- Not a preprocessor,
  code is always “valid” Verilog
- Automatically injectable
  into older code.



```
 …
/*AUTOLOGIC*/

a a (/*AUTOINST*/);
```

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

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

## Sources

- Verilator and open source design tools at http://www.veripool.org
  - Downloads
  - Bug Reporting
  - User Forums
  - News & Mailing Lists
  - These slides at http://www.veripool.org/papers/