Building Blocks for RTL Verification in ACL2

Sol Swords
Centaur Technology, Inc.

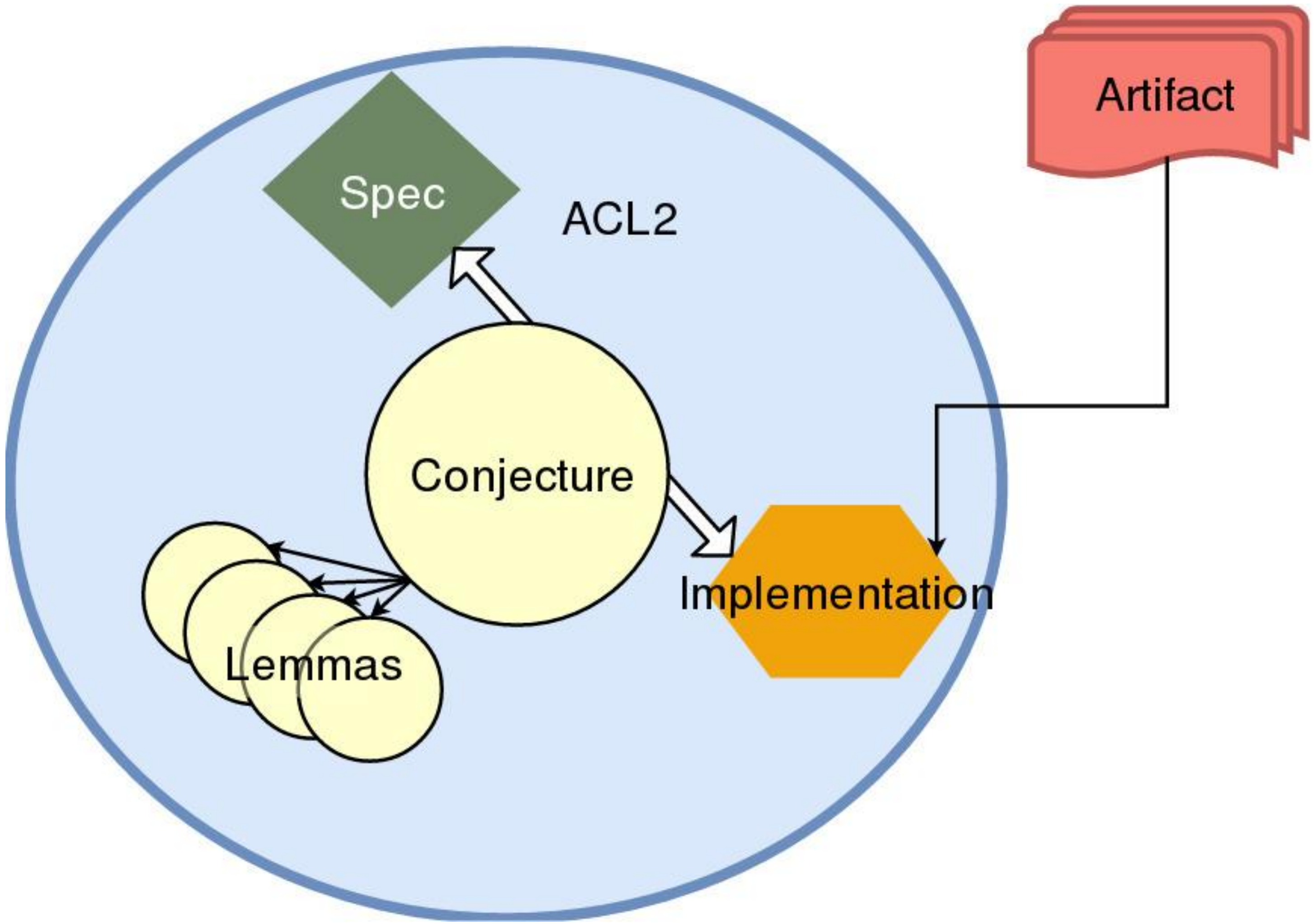
ACL2 2018

About Centaur & the FV Team

- Designers of x86 CPUs since 1995
- ~100 employees total, all in a single building in Austin:
    - Logic, circuit, verification
- FV team started in 2007
    - 4 employees
    - ACL2 based
    - Focus on datapath verification
    - Currently maintaining proofs about over 600 uops implemented on latest design.
    - Additional efforts: memory hierarchy, front-end correctness

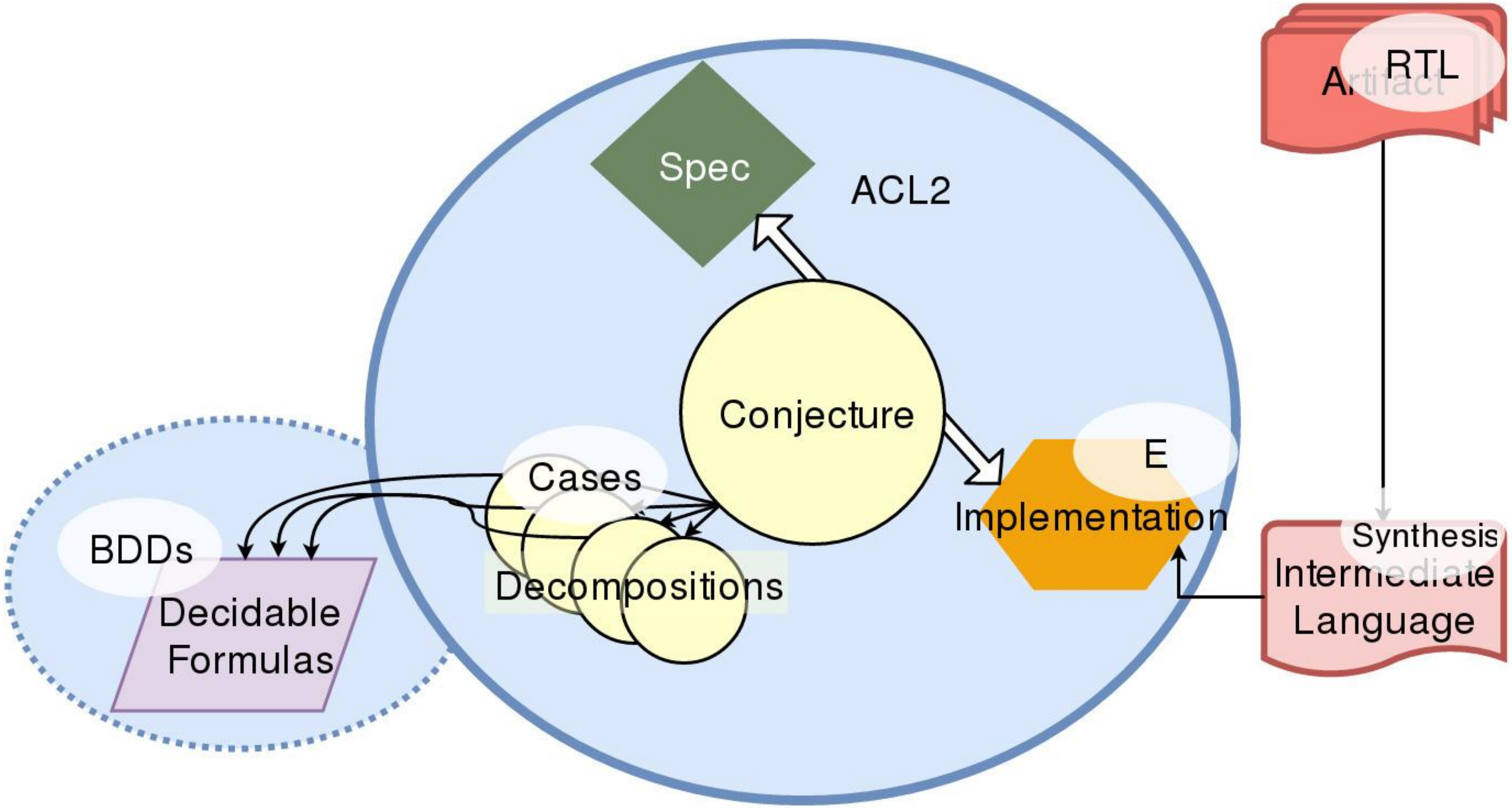
Early Centaur FV Efforts

- Divide microcode -- compute a 2N-bit (signed/unsigned) divide using native N-bit unsigned divide uop
- Implementation modeled by hand
  - Simple interpreter covering only the uops needed
  - Code transcribed into ACL2 constant by hand from sources
- Proof done using The Method


Artifact
Spec
ACL2
Conjecture
Lemmas
Implementation

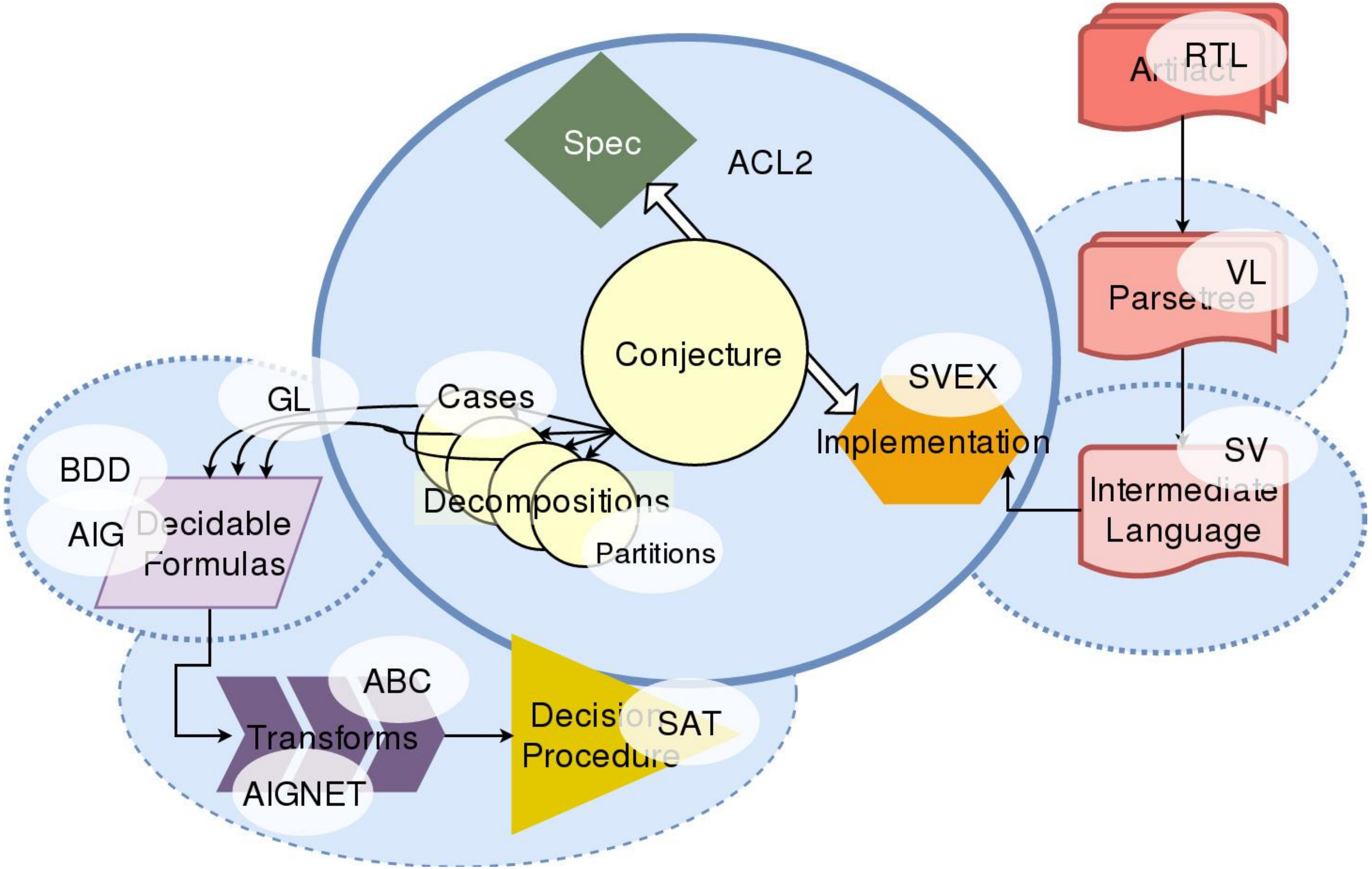
Next: FP Adder RTL

- Much bigger input artifact
- Already need much more automation
- Front end: Read in & update model of implementation automatically
  - Commercial synthesis tool dumps gate-level design
  - ACL2 parse, convert to E modules
- Back end: Proved automatically with G System (Boyer/Hunt), BDDs, split into many cases


RTL
Artifact
Spec
ACL2
Conjecture
Cases
E
Implementation
Synthesis
Intermediate Language
BDDs
Decidable Formulas
Decompositions

Since then

- “Front end” rewritten in ACL2, then rewritten another time or 2
    - VL2014/Esim -- synthesizable subset of Verilog, no more commercial synthesis
    - Reworked to support synthesizable SystemVerilog: VL/SV
- G became GL (“G in the Logic”, 2009); lots of fancy features added (rewriter -- 2013)
- Improved support for AIG → SAT flow
    - SATLINK -- plug in any modern SAT solver
    - Fraiging (SAT sweeping) and other AIGNET transforms
    - Now preferred over BDDs in most cases


RTL
Artifact
Spec
ACL2
VL
Parsetree
Conjecture
SVEX
Implementation
SV
Intermediate Language
GL
Cases
BDD
AIG
Decidable Formulas
Decompositions
Partitions
ABC
Transforms
AIGNET
Decision Procedure
SAT

Outline

- Tour through a small demo example: `bitcount` SystemVerilog module
    - Front end: VL and SV
    - Lowering proofs to the Boolean domain: GL
- Little Tricks
    - Make the spec look more like the implementation
    - Make case splits count in AIG equivalence checks
    - Make GL proofs more general

Demo example: `bitcount`

```
module bitcount #(integer logwidth = 5,
                  integer width = 1<<logwidth)
   (input clk,
    input valid,
    input [width-1:0] data,
    output resvalid,
    output [logwidth:0] count);

   generate
     genvar cycle;
     for (cycle=0; cycle<logwidth; cycle=cycle+1)
       begin : cycleblock

         .....

       end // block: cycleblock
   endgenerate

   assign count =
      ( logwidth+1 )'(cycleblock[logwidth-1].resdata);
   assign resvalid = cycleblock[logwidth-1].cyclevalid;

endmodule // bitcount
```

```
localparam integer shift = 1<<cycle;
localparam logic [width-1:0] mask
     = { width>>(cycle+1)
         { { shift {1'b0}}, { shift {1'b1}} } };
// The cycles proceed as follows:
// cycle   shift    mask (bits)     mask (hex)
//   0       1      01010101...    55555555.....
//   1       2      00110011...    33333333.....
//   2       4      00001111...    0F0F0F0F....
//   3       8                     00FF00FF....

logic [width-1:0] prevdata, indata,
          half0, half1, resdata;
logic cyclevalid, prevvalid;
if (cycle == 0) begin
  assign prevdata = data;
  assign prevvalid = valid;
end else begin
  assign prevdata = cycleblock[cycle-1].resdata;
  assign prevvalid = cycleblock[cycle-1].cyclevalid;
end

always @(posedge clk) begin
  indata <= prevvalid ? prevdata : indata;
  cyclevalid <= prevvalid;
end

assign half0 = indata & mask;
assign half1 = (indata >> shift) & mask;
assign resdata = half0 + half1;
```

```
(def-gl-thm bitcount1024-correct

  :hyp (and (unsigned-byte-p 1024 data)
            (bitp valid))

  :concl (b* ((outs (sv::svtv-run (bitcount1024-run)
                                  `((data . ,data)
                                    (valid . ,valid))))
              ((assocs resvalid count) outs))
           (and (equal resvalid valid)
                (implies (equal valid 1)
                         (equal count (logcount data)))))

  :g-bindings (gl::auto-bindings
               (:nat data 1024)
               (:nat valid 1)))
```

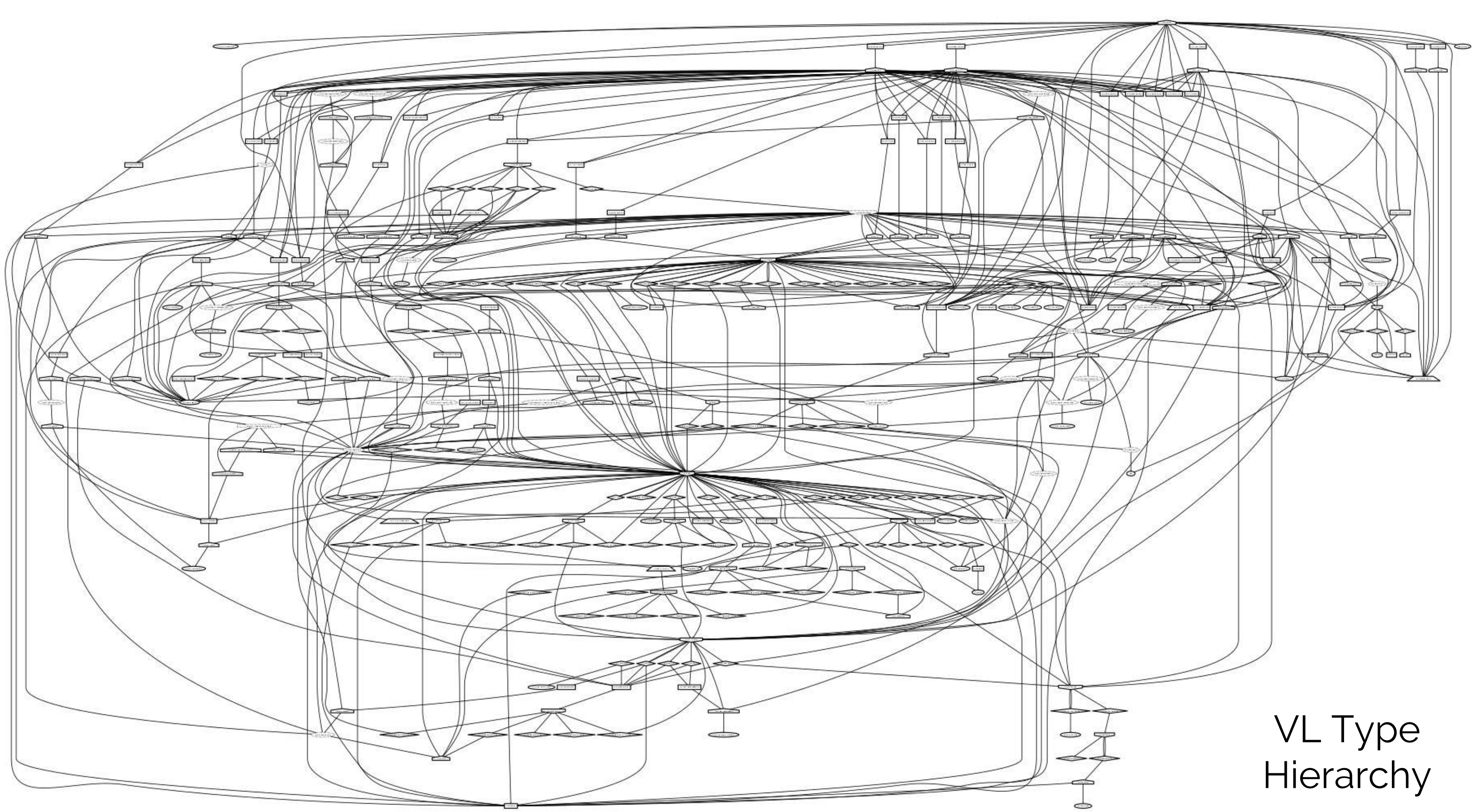
VL: SystemVerilog Parsing and Syntax Analysis

- SystemVerilog spec is 1,275 pages; “formal” syntax is 42 pages
- VL encodes parse tree using ~200 type definitions from `vl-design-p` down to `vl-bit-p`
- Tools built on VL (J. Davis):
    - Linter
    - Module browser
    - Parsetree → JSON
- “Formal semantics”: translate to SV modules ...

Parsing a SystemVerilog design

```
(defconsts (*vl-design* state)
  #!vl (b* (((mv loadres state)
             (vl-load
              (make-vl-loadconfig :start-files '("demo.sv")))))
         (mv (vl-loadresult->design loadres) state)))
```

```
(:VL-DESIGN
 ((("VL Syntax 2016-08-26"
    (:VL-MODULE
     (((("bitcount" 16777216 :VL-LOCATION . "demo.sv")
        (213909504 :VL-LOCATION . "demo.sv")
        "bitcount")
       (NIL)
       NIL
       ((:VL-ASSIGN
         ((:VL-INDEX ("count"))
          (:VL-CAST
           (:SIZE :VL-BINARY
                  (:VL-BINARY-PLUS :VL-INDEX ("logwidth"))
                  (:VL-LITERAL (:VL-CONSTINT (32 . 1) :VL-SIGNED . T))
                  ("VL_EXPLICIT_PARENS"))
           (:VL-INDEX ((("cycleblock" (:VL-BINARY (:VL-BINARY-MINUS :VL-INDEX ("logwidth"))
                                                  (:VL-LITERAL (:VL-CONSTINT (32 . 1)
                                                                             :VL-SIGNED . T))))
                         . "resdata"))))
          201326595 :VL-LOCATION . "demo.sv"))
        (:VL-ASSIGN
          ((:VL-INDEX ("resvalid"))
           (:VL-INDEX ((("cycleblock" (:VL-BINARY (:VL-BINARY-MINUS :VL-INDEX ("logwidth"))
                                                  (:VL-LITERAL (:VL-CONSTINT (32 . 1)
                                                                             :VL-SIGNED . T))))
                         . "cyclevalid")))
.....
```



VL Type Hierarchy

SV: Almost-Formal Hierarchical HDL

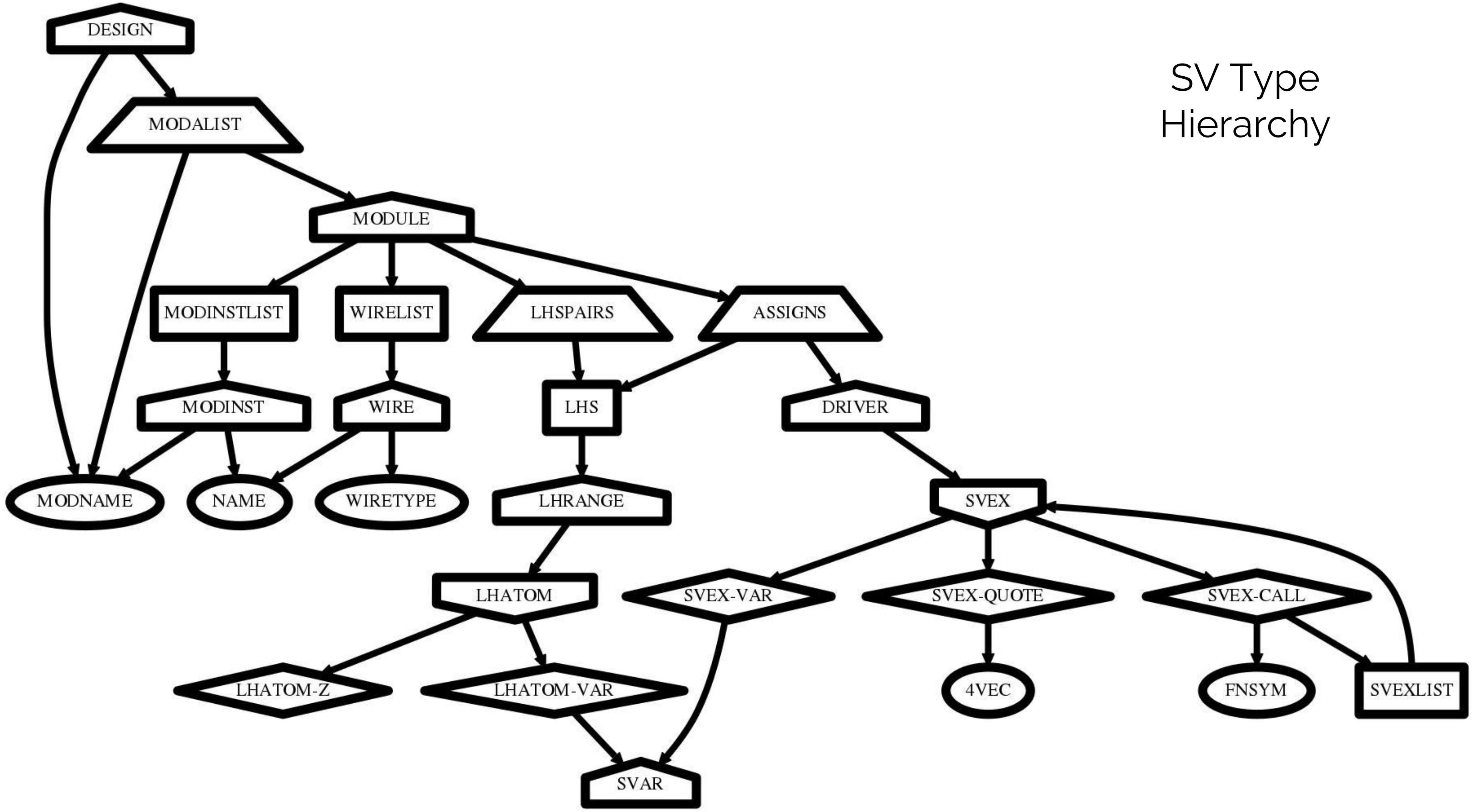
- Rich enough to be a translation target for (synthesizable) SystemVerilog
- Simple enough to someday have a formal semantics
- Every level of lexical hierarchy is expressed as a module
- No statements, only expressions assigned to parts of (wide) wires
- Simple expression format “SVEX” -- variable, quote, or operator call
- No port connections, only aliases and hierarchical references
- Sequential behavior via referencing delayed values of variables
- “Formal semantics”:
  - Flatten hierarchy and combine/split assignments to obtain one assignment for every (wide) wire
  - Compose assignments together to a fixpoint to get FSM next-states and update functions.

Translating to SV

```
(defconsts *bitcount-sv-design*
  (b* (((mv err sv-design ?good ?bad)
        (vl::vl-design->sv-design
         "bitcount"
         *vl-design*
         (vl::make-vl-simpconfig)))
       ((when err)
        (er hard? 'sv-design "~@0~%" err)))
    sv-design))
```

```
((SV::MODALIST
     ((("bitcount" :GENARRAY "cycleblock"
                   :GENBLOCK 2)
       (SV::WIRES (("prevdata" 32 . 0))
                  (("indata" 32 . 0))
                  (("half0" 32 . 0))
                  (("half1" 32 . 0))
                  (("resdata" 32 . 0))
                  (("cyclevalid" 1 . 0))
                  (("prevvalid" 1 . 0)))
       (SV::ASSIGNS (("cyclevalid")
                     (SV::? (SV::BITAND (SV::BITNOT (:VAR (:ADDRESS "clk" NIL 2) . 1))
                                        (SV::CONCAT 1 (:VAR (:ADDRESS "clk" NIL 2) . 0)
                                                    0))
                            (SV::CONCAT 1 (:VAR "prevvalid" . 1)
                                        (SV::RSH 1 (:VAR "cyclevalid" . 1)))
                            (:VAR "cyclevalid" . 1))
                     . 6)
                    (((32 . "indata"))
                     (SV::? (SV::BITAND (SV::BITNOT (:VAR (:ADDRESS "clk" NIL 2) . 1))
                                        (SV::CONCAT 1 (:VAR (:ADDRESS "clk" NIL 2) . 0)
                                                    0))
                            (SV::CONCAT 32
                                        (SV::? (SV::CONCAT 1 (:VAR "prevvalid" . 1) 0)
                                               (:VAR "prevdata" . 1)
                                               (:VAR "indata" . 1))
  .....
```

SV Type Hierarchy



Semantics for a finite run

- `Defsvtv` and derivatives compute symbolic formulas for certain signals after finite steps
- Specify (concrete or symbolic) inputs, delays, outputs to extract
- Flattens hierarchy, composes expessions for a single time slice, unrolls phases
- Result: essentially a mapping: output signals → SVEX expressions

```
(sv::defsvtv-phasewise bitcount-run
  :design *bitcount-sv-design*
  :phases ((:inputs (("clk" 0 :toggle 1)
                     ("valid" valid)
                     ("data" data)))
           (:delay 10  ;; 10 phases = 5 clk cycles
            :outputs (("count" count)
                      ("resvalid" resvalid)))))
```

Simulation of a finite run

```
(sv::svtv-run (bitcount-run)
              '((data . #x38f0a500)
                (valid . 1)))
```

⇒

```
((COUNT . 11) (RESVALID . 1))
```

Proofs about SVTVs

- You could try opening up definitions & do proofs using The Method
  - I don’t recommend this
- Usual route: bit blast using GL
  - Intermediate form: convert SVEX objects to vectors of AIGs
  - Oftentimes, just prove it equivalent to a spec & be done
  - Sometimes split into cases
  - Sometimes decompose by proving lemmas about internal signals

GL Background

- Stands for “G in the Logic,” based on Boyer & Hunt’s “G system”
- Idea: Represent ACL2 conjecture as Boolean function (AIG, BDD)
- Express ACL2 objects containing symbolic Booleans and integers
- Operate on these objects using symbolic analogues of functions, e.g.:

```
(equal (+ (gl-obj-eval a env) (gl-obj-eval b env))
       (gl-obj-eval (gl-obj-+ a b) env))
```

- Symbolically interpret terms:

```
(equal (gl-obj-eval (gl-term-interp my-term bindings) env)
       (term-eval my-term (gl-obj-alist-eval bindings env))*
```

* (Some technical details elided…)

```
(def-gl-thm bitcount1024-correct

  :hyp (and (unsigned-byte-p 1024 data)
            (bitp valid))

  :concl (b* ((outs (sv::svtv-run (bitcount1024-run)
                                  `((data . ,data)
                                    (valid . ,valid))))
               ((assocs resvalid count) outs))
            (and (equal resvalid valid)
                 (implies (equal valid 1)
                          (equal count (logcount data)))))

  :g-bindings (gl::auto-bindings
                (:nat data 1024)
                (:nat valid 1)))
```

GL Scaling

- Default: use BDDs --
- 128 bits: 2 sec
- 256 bits: 14 sec
- 512 bits: 95 sec
- Trend: 1024 > 10 minutes (too impatient!)
- Switching to SAT or FRAIGing + SAT → even worse (for this problem)
- How do we get there? Back to theorem proving?

Little Tricks

Trick 1: Make the spec like the implementation

- Don’t read the RTL more than necessary…
- But the algorithm is basically just summing the bits in a tree
- `Logcount` sums them in a linear scan:

```
(defun logcount-of-natural (n)
  (if (zp n)
      0
    (+ (if (logbitp 0 n) 1 0)
       (logcount-of-natural (ash n -1)))))
```

- Maybe if we represent `logcount` as a more treelike sum...

```
(define logcount-rec ((logwidth natp) (x integerp))
  (if (zp logwidth)
      (loghead 1 x)
    (+ (logcount-rec (1- logwidth) x)
       (logcount-rec (1- logwidth) (logtail (ash 1 (1- logwidth)) x))))
  ///
  (defthm logcount-rec-is-logcount
    (equal (logcount-rec logwidth x)
           (logcount (loghead (ash 1 (nfix logwidth)) x)))))

  (gl::def-gl-rewrite logcount-of-u1024
    (implies (unsigned-byte-p 1024 x)
             (equal (logcount x)
                    (logcount-rec 10 x))))
```

Close enough?

- SAT solving now solves 1024-bit case in 2.7 sec
- FRAIGing: 0.03 sec

Fraiging is very fast at comparing very similar formulas!

Relies on finding many exact functional matches between AIG nodes.

→ Trick 1: Prove your spec equivalent to something close to the implementation algorithm.

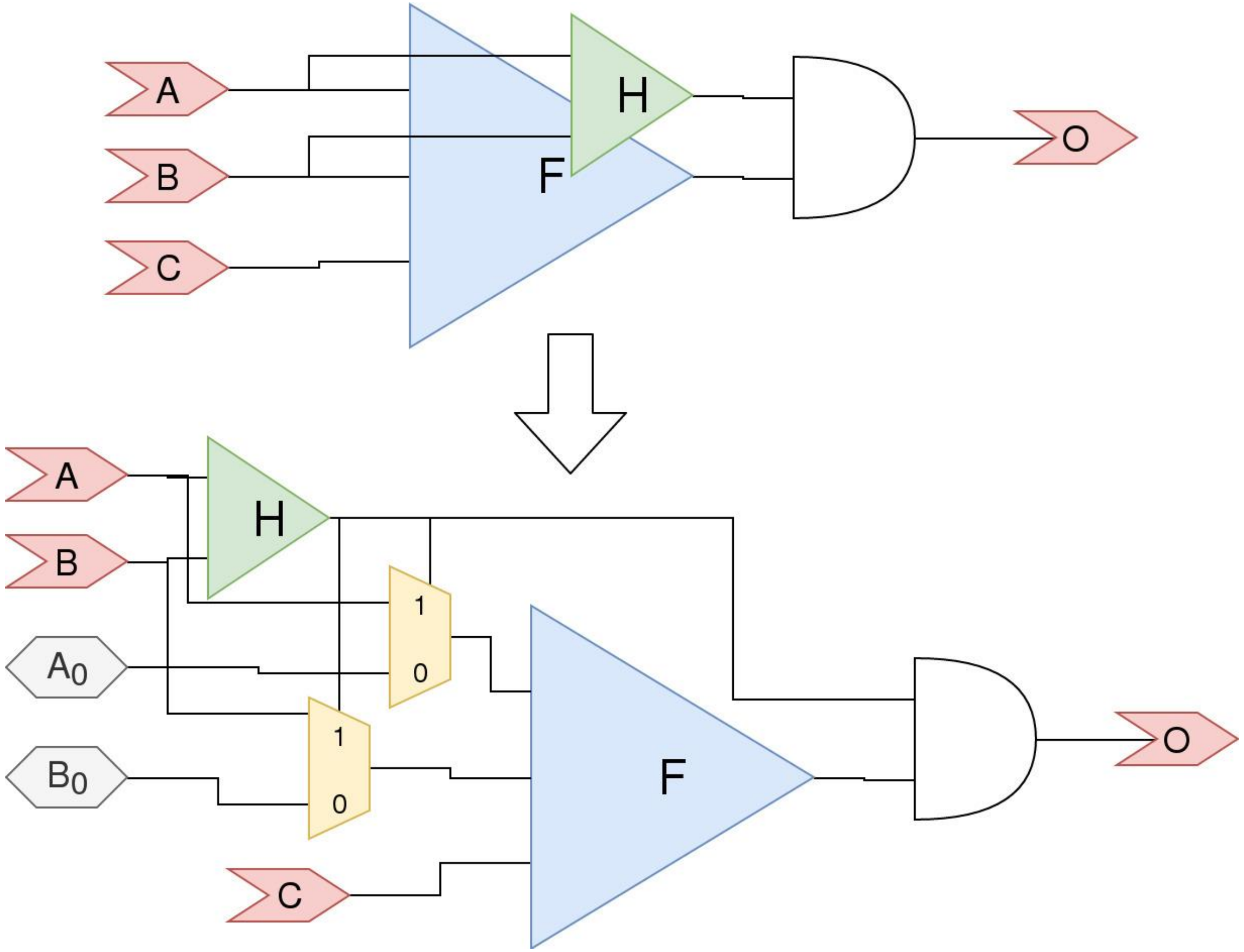
Trick 2: Make case splits count

- Fraiging only works if nodes have EXACTLY equivalent function
- Sometimes they're equivalent EXCEPT in unimportant cases → Fraiging fails
- Example: FP addition paths: either normalize or round, depending on exponents and signs
  - If exponents within 1 and signs opposite (effective subtract), normalize
  - Otherwise round
- Don't know what normalization path does when we're going to round
- But if the spec does something different, then there won't be full equivalences between the nodes!

A simple transformation can fix this...

Observability Transform

- Transform the inputs to the main formula so that they always satisfy the “hyp”
    - Mux: if hyp is satisfied, then original inputs, else arbitrary satisfying assignment
- Equivalent-under-hyp nodes within main formula are then fully equivalent
- Effects on performance:
    - Random simulation is less useful
    - Counterexample simulations are more important
- Transform available in aignet package
    - Configurable, but usually just works


A
B
C
F
H
O
A
B
H
1
0
1
0
A_0
B_0
F
O
C

Observability + Fraiging results

- Extended precision FP Add proved equivalent to spec in ~4 minutes
    - Four cases: Special, normalize, round effective add, round effective subtract
- For comparison, our previous result with BDDs (*Use of Formal Verification at Centaur Technology*, 2010): 48 minutes, >800 cases
- Intel result (*Universal Boolean Functional Vectors*, 2015): double precision add verified in 30 minutes
    - No case splitting using BDD-based functional parametrization approach
    - Their previous result: 51.5 hours, 231 cases

Trick 3: Make GL Proofs More General

GL proofs are commonly written with rigid *shape specs* binding every free variable:

```
:g-bindings (gl::auto-bindings
             (:nat data 1024)
             (:nat valid 1))
```

These have some disadvantages:

- Require otherwise unnecessary hyps

```
:hyp (and (unsigned-byte-p 1024 data)
          (bitp valid))
```

- Lead to overly specific objects, making composition difficult

```
`((data . ,data)
  (valid . ,valid))
```

- In examples with too many variables, may be impractical to build shape specs or verify coverage

Ditching Shape Specs for Free Variables

- Decide on a set of accessors sufficient to extract relevant bits from your variables
- Make them uninterpreted (prevent function expansion) using `gl-set-uninterpreted`
- Prove any rewrite rules necessary
- Provide counterexample extraction rules for rebuilding objects from Boolean assignments
- Lots of work, but perhaps can just do it once and use it for lots of proofs…
- Useful GL rewriting theory for SV provided in `centaur/sv/svex/gl-rules`

With suitable GL rewrite rules (i.e. “centaur/sv/svex/gl-rules”)…

```
(def-gl-thm bitcount1024-correct3
  :hyp (and (integerp (sv::svex-env-lookup 'data env))
            (integerp (sv::svex-env-lookup 'valid env)))
  :concl (b* ((outs (sv::svtv-run (bitcount1024-run) env
                                  :boolvars nil :allvars t))
              ((assocs resvalid count) outs)
              (data (loghead 1024 (sv::svex-env-lookup 'data env)))
              (valid (loghead 1 (sv::svex-env-lookup 'valid env))))
           (and (equal resvalid valid)
                (implies (equal valid 1)
                         (equal count (logcount data)))))
  :g-bindings nil)
```

Final Thoughts

- VL/SV show that big, ugly languages can be handled, “formalized” (given time & commitment)
    - Helpful to have a way to ramp up -- i.e. code that doesn’t use too many features
    - Some benefit to working directly from sources vs. using 3rd party tools to simplify
- Boolean equivalence checking is quite a hammer
    - (Granted, not everything is a nail.)
    - Several alternatives if it can’t get your whole proof:
        - Split into cases (and maybe use observability xform to make conditional equivalences useful)
        - Split spec/implementation into subparts
        - Lift implementation to something that can be proven equivalent to spec (or lower spec to something that can be proven equivalent to implementation)